第 1 章～第 5 章の内容は一部、TX19 には当てはまらないものがあります。それらは TX19 アセンブラ固有の仕様とする部分であり、第 6 章を参照して下さい。

第1章 ソースプログラム

1.1 ソースプログラムの構造

機械命令を記述したソースステートメントを単に並べただけではプログラムにはなりません。プログラムには、ある一定の構造が必要となります。
以下では、プログラムの構造に関して説明を行います。

1.1.1 モジュールとセクション

プログラムの機能分割　アセンブリ言語のプログラミングの基本となる概念は、モジュールとセクションです。

モジュール　大規模なプログラムを作成する際に、そのプログラム全体を1つのソースファイルとして作成するよりも、機能別に複数のファイルに分割して、それらを後から結合してまとめたほうが、デバッグ、修正、編集がしやすくなりプログラム開発の効率も上がります。一度作成したプログラムを別のプログラム作成で再利用することも可能となります。プログラムを分割して作成した場合の、各々のソースファイルをモジュールと呼びます。(1ファイル1モジュール)

セクション　また、モジュールは複数のセクションで構成されています。セクションは、プログラムをメモリ上に配置する場合の単位となります。セクションを機械命令のセクションやデータのセクションなどに分けて指定することで、ソースプログラムのそれぞれの部分をメモリ領域のどこに配置するかを細かく指定することができます。例えば、ソースプログラムを機械命令、変数、定数などに分類をして作成すると、開発したプログラムをROM化する場合に便利です。実際にセクションを細かく分割し、機械命令の部分や値が変更されない定数などはROMに、値の変更される変数はRAMに配置します。

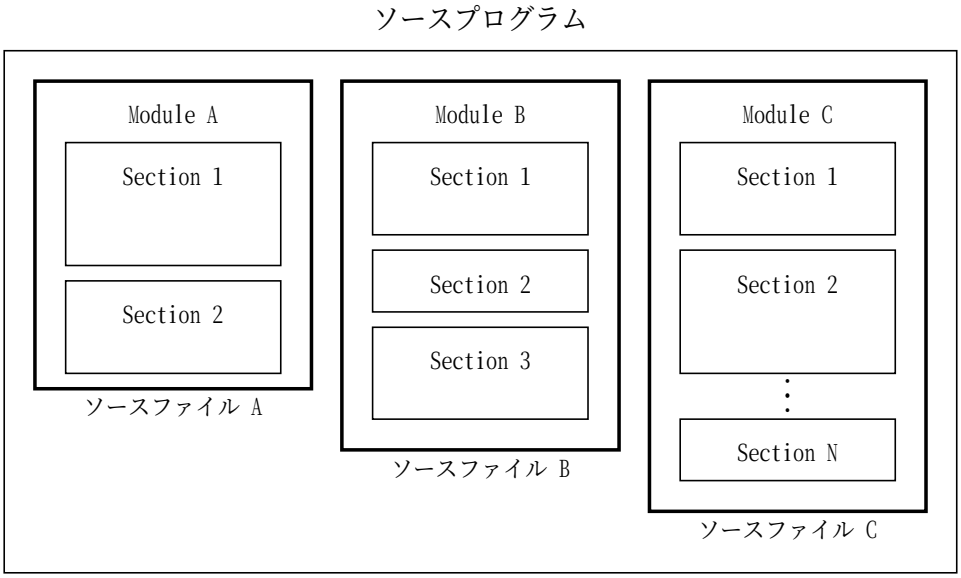


図1.1 ソースプログラムの論理的構造

図 1.1 に示すように、ソースプログラムは複数のモジュールで構成され、モジュールはいくつかのセクションによって構成されます。これらのモジュールは、それぞれアセンブラによってアセンブルされ、リンケージエディタによってセクション単位で結合、編集されて、一つの実行可能なオブジェクトモジュールになります。

1.2 モジュール

1.2.1 モジュールの構造

一つのマイコン応用システムのソースプログラムは、図 1.1 に示したように複数のモジュール(ファイル)で構成されます。一つのモジュールは通常複数のセクションで構成されますが、セクション以外に記述しなければならないものがいくつかあります。

モジュール構成例

オプション定義	-lw120 -lp60 -e errfile.err	←必要があれば固定オプションを記述することも可
コメント	; XXXXX model A control program ; Apr.1,1999 by Watanabe	←必要に応じて随時コメント文を挿入
基本制御文	$maximum	←アセンブラによって基本制御文がないものもある
module擬似命令	module main_asm	←moduleはオプション、基本制御文の後
一般制御文	$title "XXXXX Main control" $nolist $include "define.inc" $list	←必要に応じて一般制御文を挿入 (moduleとendの間)
シンボル定義	AAA equ 0x10 BBBB equ -1 CCC equ 0x8000 $eject	←#define,define(),equ等
セクション定義	; Data Area for Timer timer_data section data small time1 dsl 1 ;time data counter dsw 1 ;counter : ; Timer Control Routine timer_prog section code medium start: ldw (counter),CCC loop1: cp wa,(counter) :	←データセクションデータ定義部 ←コードセクションプログラム部
end擬似命令	end	

図1.2 モジュールの構成例

図 1.2 は標準的なモジュールの構成です。プログラムによっては必要のない項目もありますので、あくまでも目安として参考にして下さい。

1.2.2 モジュールの定義

図 1.2 に示したように、モジュールは module 擬似命令で定義します。

【記述形式】

```
module   <モジュール名>
```

<モジュール名>はシンボルの記述規則に従った名前を指定します。各ソースファイルに一つモジュール名を定義します。モジュール名は各ファイル固有でなければなりません。

1.3 セクション

セクションはリンケージエディタの結合処理の対象となる最小単位で、実行時のメモリ上の配置の単位となるものです。メモリ上に連続して配置したい場合は、同じセクション内にプログラムまたはデータを記述します。プログラムはプログラム、データはデータとして一つのセクションにまとめます。プログラムとデータは種類が違いますので、同じセクションにまとめることはできません。

セクションの定義

図1.2に示したように、セクションはsection擬似命令で定義します。

【記述形式】

```
<セクション名>  section  <種類>  [<配置>]  [<アライメント>]
```

<セクション名>はシンボルの記述規則に従った名前を指定します。セクション名が同じであれば、たとえ別ファイルに記述されていても、そのセクションは連続して配置されます。同一のセクション名で異なる<種類>や<配置>を指定することはできません。

1.3.1 セクションの種類

上記の記述形式において、<種類>はセクションの種類を示す属性です。セクションの種類は次の 3 種類があり、大文字、小文字の区別はありません。<種類>の指定は省略できません。

code　data　romdata

CODEセクション

CODEセクションは機械命令を記述するセクションです。オブジェクトコードが出力されます。

```
code_1 section  code          ; relocatable code section
waitff:
       call     time_set      ; location address is fixed
                              ; after linking
loop:
      ldw       wa,(port1)
       cp       wa,0xff
       jr       z,loop
       set      0,(effect_flag)
       ret
```

DATAセクション　DATAセクションはプログラム実行時に値が変更される変数を記述するセクションです。通常RAMのデータ領域で、初期値を持ちません。dsb,dsw,dsp,dsl擬似命令で領域を確保します。このセクションはオブジェクトコードを持たず、領域のみが確保されます。

```
ioport section data abs=0x0 ; start address 0x0
port0  dsb  1    ; port 0, like 1 byte data area
port1  dsb  1    ; port 1
p0cr   dsb  1    ; control register for port 0
       dsb  1
p1cr   dsb  1    ; control register for port 1
```

ROMDATAセクション　ROMDATAセクションはプログラム実行時に値が変更されない定数を記述するセクションです。通常ROMのデータ領域で、初期値を持ちます。db, dw, dp, dl, dfb, dfw, dfp, dfl, fdatas, fdatad, fdatax 擬似命令で定数データを定義します。初期値データがオブジェクトコードとして出力されます。

```
ioport  section romdata large; relocatable romdata section
charac1 dw      0x2f10       ; 2 byte ROM data
charac2 dw      0x51c8
pattern dl      0xffffffff   ; 4 byte ROM data
```

1.3.2 セクションの配置属性

<配置>は、セクションの配置属性です。配置属性にはリロケータブル指定とアブソリュート指定の 2 種類があります。

リロケータブル指定　セクションを宣言するときに、セクションの配置属性に配置領域を指定した場合や、配置属性を省略した場合には、リロケータブル(相対番地)セクションの指定となります。配置領域は、large, medium, smallのいずれかを指定します。相対番地セクション内で定義したシンボルには、セクション内での相対的なアドレスが割り当てられます。リンケージエディタでセクションを結合・メモリ配置した後、セクションの先頭番地が決定されます。このとき先頭番地は<配置>に続く<アライメント>「align=値」で指定した値の倍数に調整されます。

リロケータブルセクションの利点はリンク時に配置指定が自由にできることです。設計の変更や Bug 修正などによって影響を受けないのでいちいちソースプログラムのアドレス指定を変更する必要がなくなります。すなわち、セクション配置が変わったときに section 擬似命令を書き換えずにすみます。このため、通常セクションはリロケータブル指定にします。

次のリスト例で、IDT1 セクションがリロケータブル指定で、18,19,20 行目のアドレス表示はセクションの頭をゼロとした相対値になっています。また、IDT1 セクション内のシンボル loc_2 はリロケータブルシンボルなので、これを参照している 23 行目のオブジェクトは仮に 00000000 となっています。これらはリンク後に値が決まります。

リスト例

```
                        +0 17  17 IDT1    section data  large
00000000                +0 18  18 loc_1    dsb     1
00000001                +0 19  19 loc_2    dsw     1
00000003                +0 20  20          dsb     1
                        +0 21  21 ICD3     section code medium
00000000 330000 R       +0 22  22 label1:  ld      hl,pub_1
00000003 4000000000 R   +0 23  23          ld      xwa,loc_2
00000008 10             +0 24  24          rcf
00000009 9093           +0 25  25 label2:  adc     hl,(xwa)
```

アブソリュート指定

セクションを宣言するときに、セクションの配置属性に「abs=アドレス値」と指定した場合、アブソリュート(絶対番地)セクションとなります。配置アドレスにはセクションの先頭番地を指定します。指定した配置アドレス値は、アセンブル時にオブジェクトファイルにそのまま出力されます。この値は、リンク処理の際にも変更されることはありません。

リスト例

```
                +0  13   13    IDT0    section data    abs=0x80
00000080        +0  14   14    pub_1   dsb      6
00000086        +0  15   15    pub_2   dsl      1
```

絶対番地セクション内で定義したシンボルには配置アドレスで指定した値を基準とした固定のアドレスが割り当てられます。このセクションは、リンケージエディタによって他のセクションと結合されることはありません。配置アドレスも変更されません。

1.4 シンボルの定義と参照

通常プログラムのコーディングで、ある数値やロケーション(アドレス)を直接記述するかわりに、シンボルを使用します。プログラム中で参照するシンボルには以下のような種類があります。

シンボルの種類

- ラベル
  機械命令コードのロケーション(アドレス)を表します。定義時には ":" (コロン)をつけ、分岐命令などで参照されます。
- 変数名
  データを取り扱うために定義するデータ領域の名前です。データ定義擬似命令で定義し、ラベルと違って ":" はつけません。データを取り扱うほとんどの命令で参照されます。
- EQUシンボル
  数値を直接記述しなくてもいいよう、EQU擬似命令で定義されたシンボルです。

1.4.1 モジュール間のシンボルの参照

シンボルの定義と参照が一つのモジュール内でのみ行われている場合は、アセンブラはそのシンボルが持つ各種情報を知ることができ、正常にアセンブルできます。しかし、ソースプログラムを複数のモジュールで構成する場合、モジュール間のシンボルの相互参照が発生します。アセンブルはモジュール単位で行われるため、モジュール間参照には特別な情報受け渡し手順が必要になります。

シンボルの外部宣言

public

あるモジュール内で定義したシンボルを別のモジュールから参照できるように宣言することを外部宣言と呼び、その宣言されたシンボルを外部定義シンボルといいます。

シンボルを外部定義シンボルとして宣言するためには、"public"擬似命令を用います。

```
public  シンボル名
```

シンボルの外部参照

extern　別のモジュール中で定義されているシンボルを参照することを外部参照と呼び、そのシンボルを外部参照シンボルといいます。外部参照シンボルは、アセンブル時には値が決定されず、リンケージエディタによってセクションが結合、編集される際に値が決定されます。

シンボルを外部参照シンボルとして宣言するためには、"extern"擬似命令を用います。

```
extern [サイズ] シンボル名
```

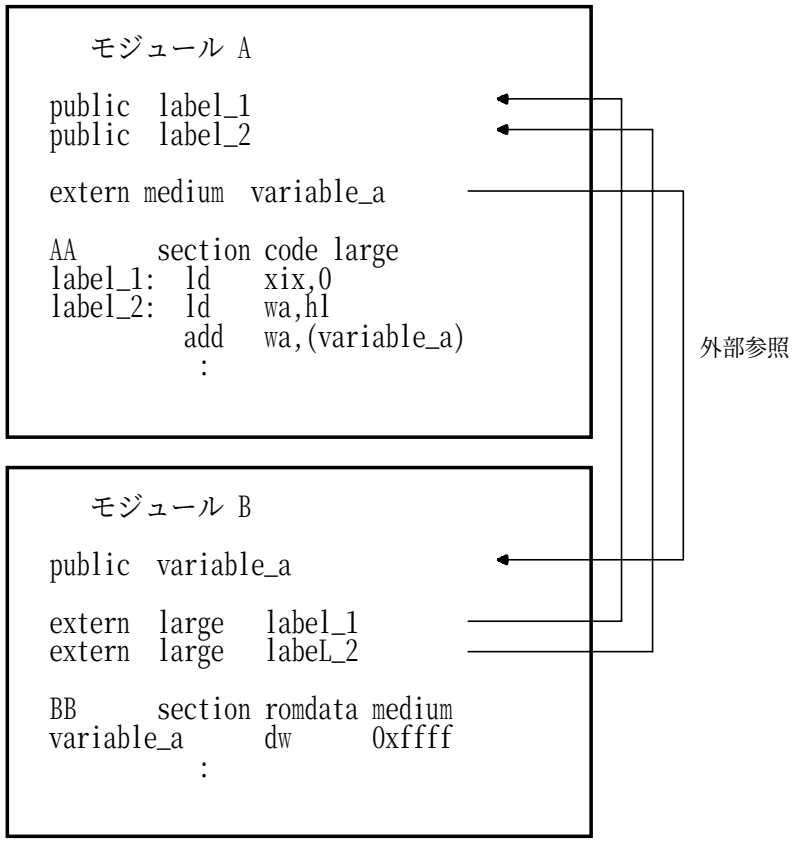


図2.3 シンボルの外部宣言と外部参照

1.4.2 モジュール内のシンボルの参照

ローカルシンボル　モジュール間で相互参照が行われず、1モジュールの中で定義と参照が閉じている場合、このシンボルをローカルシンボルといいます。ローカルシンボルに対しては特別な宣言は必要ありません。

前方参照と後方参照　1つのモジュール内でのシンボルの参照は、シンボルを参照している命令とシンボルの定義の相対的な位置関係により、前方参照と後方参照に分類することができます。

前方参照

ソースファイルにおいて、シンボルを参照している位置よりも前方(ソースファイルの最終行の方向)にシンボルの定義が存在する場合の参照を前方参照といいます。

後方参照

ソースファイルにおいて、シンボルを参照している位置よりも後方(ソースファイルの先頭行の方向)にシンボルの定義が存在する場合の参照を後方参照といいます。

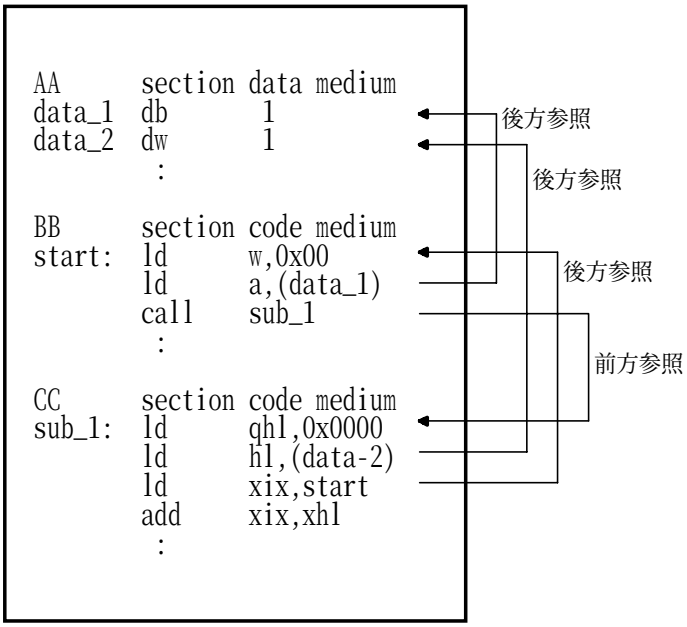


図2.4 シンボルの前方参照と後方参照

アセンブラは、ソースファイルを先頭行から順番に読み込んでアセンブル処理を行います。そのため、後方参照のシンボルに対しては、ソースファイルを 1 度読み込むだけで値を決定することができます。一方、前方参照のシンボルに対しては、ソースプログラムの 1 度目の読み込みでは値の決定を保留し、2 度目の読み込みで値を決定します。

1.4.3 シンボルの値

アセンブラは、それぞれのシンボルの値を決定しますが、その値の決定方法により、相対値と絶対値に分類されます。

相対値　相対セクション内で定義したラベルに割り当てられる値で、セクションの先頭からの相対的なアドレスが割り当てられます。この値はリンク処理の際に調整されます。

絶対値　equ擬似命令で定義したシンボルや絶対セクション内で定義したラベル、変数名に割り当てられる値です。この値はリンク処理でも変更されません。

1.5 ソースステートメント

図 1.2 中のプログラムの１行１行がソースステートメントです。それぞれのソースステートメントはアセンブラが解釈できるように、決められたフォーマットで記述する必要があります。

ステートメントの単位

アセンブリ言語のソースプログラムでは、ソースプログラムの各行がソースプログラムの最小単位であり、通常は1行に1つの命令を記述します。この最小単位を"ソースステートメント"と呼びます。

ソースステートメントには機械命令や擬似命令などの様々な要素を記述します。これらの要素はソースステートメントのある決まった場所、「フィールド」に記述しなければ、正しい意味を持ちません。

以下では、ソースステートメントの種類と記述方法について説明します。

1.5.1 ソースステートメントの種類

アセンブリ言語のソースプログラムでは、各行(ソースステートメント)を決められたフォーマットで記述する必要があります。ソースステートメントは、以下のような構造で記述します。

記述形式

```
[ <ラベル>: ] [ <オペレーション>  <オペランド> ] [ ;<コメント> ]
```

それぞれの構成要素を「フィールド」と呼びます。ソースステートメントはどのフィールドが使われているかで、命令行、コメント行、空白行の 3 つに分類することができます。各フィールドは必要がなければ省略できます。左より、ラベル、オペレーション、オペランド、コメントの順に並んでいれば、どのフィールドが、何カラム目から始まっても構いません。

1.5.2 命令行

ラベルフィールドにラベルを記述した行や、オペレーションフィールドに機械命令、擬似命令、制御命令を記述した行を命令行と呼びます。

一般的な命令行の例で説明します。

命令行の例

```
lab1: ld wa,bc ; LOAD the content of BC register to WA register.
```

このステートメントには、全てのフィールドに要素が入っています。それぞれのフィールドの要素は、

ラベルフィールド	lab1
オペレーションフィールド	ld
オペランドフィールド	wa,bc
コメントフィールド	LOAD the content of BC register to WA register.

となります。

各フィールドは、タブまたは空白文字で区切る必要があります。オペランドフィールドは、コンマ","で区切って複数のオペランドを記述します。
各フィールドはそれぞれ規則に従って省略することができます。ラベルフィールドは必要がなければ省略できます。また、コメントフィールドも省略可能です。

省略形

```
Labela: nop    ; オペランドフィールドがない場合(命令によって異なる)
               ; 次行はラベルフィールドとコメントフィールドを省略し
               ; た場合
       ld  wa,bc
labelb:        ; ラベルフィールドのみの場合
```

命令行の種類　ソースステートメントのうち、命令行のみがプログラムとしての意味を持ちます。次に説明する空白行およびコメント行はアセンブラによって無視されます。
命令行は、オペレーションフィールドに、ターゲット CPU の機械命令を記述するか、アセンブラの擬似命令、制御命令を記述するかで、機械命令行、擬似命令行、制御命令行に分類することができます。

- ❑ 機械命令行　ターゲットCPUの機械命令を記述した行
- ❑ 擬似命令行　アセンブラの擬似命令を記述した行
- ❑ 制御命令行　アセンブラの制御命令を記述した行
  命令の先頭にドル記号($)が付く

機械命令、擬似命令、制御命令の詳細については、「第2章 アセンブラの基本文法」以降を参照してください。

1.5.3 空白行とコメント行

空白行とコメント行はアセンブラにとっては意味を持ちません。ただし、プログラマにとっては、プログラムを見やすくする、仕様や覚書きを記述するなど、重宝する行です。

空白行　全てのフィールドを省略した行を空白行と呼びます。アセンブラに対しては何の意味も持ちません。ソースプログラムを見やすくするときなどに使用します。

コメント行　コメントフィールドのみの行をコメント行と呼びます。アセンブル時にはコメント行は無視されます。

1.5.4 ソースステートメントのフィールド

ラベルフィールド

ラベルまたは変数名を記述する位置をラベルフィールドといいます。通常、ラベルフィールドは物理行の先頭から始まり、ラベル定義時にはコロン":"を付加します。ラベルフィールドにはシンボルを記述します。シンボルは、通常は英字で始まる英数字文字列です。詳細については、「2.3 識別子」を参照してください。
ラベルフィールドにラベルを記述した場合は、ラベルフィールドのみの行も行として許されます。

ラベルフィールド例

```
LABEL:                          ; ラベルのみ
Label:    ld    wa , bc         ; ラベル + 機械命令
VARIABLE                        ; 変数名のみは不可  !! error !!
Variable  dsw   1               ; 変数名 + 擬似命令
```

オペレーションフィールド

オペレーションフィールドは、プログラムの動作を指定する命令を記述するフィールドです。オペレーションフィールドには以下のものを記述することができます。

- ❑ アセンブラの機械命令 (ニモニック)
- ❑ アセンブラの擬似命令
- ❑ アセンブラの制御命令

ラベルがある命令行では、ラベルフィールドの後ろに空白文字またはタブを入れてオペレーションフィールドを記述します。

ｵﾍﾟﾚｰｼｮﾝﾌｨｰﾙﾄﾞ例

```
Label:    ld   wa,10
          add  bc,wa
          nop          ; This is "No Operation" instruction
```

1 行目は、ラベルフィールドに続けてオペレーションフィールドを記述した例です。2 行目は、機械命令のみを記述した例です。ラベルフィールドはありません。3 行目は、オペレーションフィールドにコメントを続けて記述した例です。

オペランドフィールド

オペランドフィールドは、オペレーションフィールドに記述した機械命令や擬似命令の処理する対象である、オペランドを記述するフィールドです。例えば、

```
label:  ld    wa , bc
var1    db    0xff
```

上記の場合は、"wa", "bc", "0xff"がオペランドです。オペランドの個数はオペレーションフィールドに書かれた命令によって決まります。オペランドを必要としない命令では、オペランドフィールドは存在しません。また、オペランドフィールドのみの行は存在しません。必ずオペレーションフィールドとペアになります。
オペランドフィールドには以下のものを記述することができます(個々のオペランドの説明は後述します)。

- 数値(定数)
- シンボル
- レジスタ、フラグなど
- 特殊オペランド

オペランドフィールドは、オペレーションフィールドと 1 つ以上のタブまたは空白文字で区切ります。オペランドフィールドに複数のオペランドを指定する場合には、コンマまたは空白で区切ります。コンマか空白かはオペレーションフィールドの命令によって決まります。コンマの前後にはタブまたは空白文字を入ることができますが、なくてもかまいません。

正しい例

```
Ld  wa,10       ; これらのステートメントはどれも10という値をwaという
Ld  wa,   10    ; レジスタに転送する機械命令です
Ld  wa   ,   10 ; カンマの前後に空白を挿入可能
```

これらの 3 つのソースステートメントは同じ内容を表しています。どのソースステートメントにもオペレーションフィールドの後に 1 つ以上の空白文字があり、その後にオペランドフィールドを記述しています。複数のオペランドの間を区切るカンマ(,)の前後には、空白文字があってもなくても構いません。

誤った例

```
Ldwa,10 ; オペレーションフィールドとオペランドフィールドが区別
        ; できない
        ; ldとwaの間に空白行が必要
ld wa 10; 機械命令ではオペランドは','で区切らなければならない
```

コメントフィールド

コメントフィールドは、セミコロン(;)から行の終わりまでです。コメントフィールドに記述した内容はアセンブル時には無視されるので、プログラムの注釈を記述するなど自由に使用することができます。必要のない場合には省略が可能です。

- コメントフィールドはセミコロン(;)で始まり、改行で終わります。
- コメントフィールドだけの行を記述することもできます。その場合は、行の先頭を";"で始めて行全体をコメントフィールドにします。
- 複数の行にわたってコメントを記述する場合には、改行の後はふたたび";"で始めてコメントを記述します。
- コメントフィールドの中では、ソースステートメントの中で使える文字セット以外の任意の文字を使うことができます(日本語でコメントを記述することもできます)。
- オペランドフィールドのダブルクォーテーション(")またはシングルクォーテーション(')で囲まれた文字列の中のセミコロン(;)は、コメントフィールドを示す記号としては扱われず、通常の文字として扱われます。
- コメント記号が2個連続した場合(ダブルコメント記号)は、ダブルコメント記号以降のその行の記述は、アセンブルリスト上には現われません。

正しい例

```
Count   equ   10          ; count= カウンタの初期値
COUNT   equ   10          ; COUNT= カウンタの初期値
Label:  ld    wa,bc       ; このようにいくつもの行にわたった
                          ; コメントを記述することもできます
;これはコメント行の例です
```

例の最後の行のように行の先頭からコメントを記述することもできます。また、複数の行にわたってコメントを記述する場合にも、改行後は";"でコメントフィールドを指定します。

誤った例

```
COUNT   equ  10              COUNT=カウンタの初期値
        db   "abc;def;ghi"   ; 文字列中の;は通常の文字として扱われる
                             ; defやghi"はコメントにはならない
        dw    0x1234         ; ここはコメント
                              コメントではない
```

1.5.5 ステートメントの終了

MS-DOS vs UNIX

アセンブラは、ステートメント単位で処理を行ないます。1つのステートメントは、MS-DOSファイルでは通常<CR><LF>で、UNIXファイルでは<LF>で終了します。例えば、MS-DOSで作成したソースファイルをUNIX上でアセンブルするような場合は、<CR>の削除や漢字コードの変換など注意が必要です。

1.5.6 オプション指定部

アセンブラ起動時に指定するオプションパラメータをソースファイルの先頭に記述できます。

パラメータ指定部は、必ずソースファイルの先頭に記述します。パラメータ指定部より前には何も(コメント行や空白行も)記述できません。また、パラメータ指定部は複数行にまたがることが出来ます。

パラメータ指定部には、"-f", "-H", "-V", "-$"以外のオプションが記述できます。オプションについての詳細は「C,C-Like,アセンブラ取扱説明書[操作編]」を参照して下さい。

```
-lw120
-o outfile.abs

$MAXIMUM
cd_sec  section code    medium
start:  ld wa,(memory)
       :
       :
```

第2章 アセンブラの基本文法

ソースステートメントは、ニモニックやレジスタ名、数値データや文字列データなどの要素を使って記述します。これらの要素はすべて文字の並びで表され、一定の表記法に従って記述しなければなりません。

本章では、アセンブラのソースステートメントを構成している要素を説明します。

2.1 文字セット

アセンブラでソースプログラムを記述するときに、ソースステートメントに記述できる文字を表 2.1 に示します。

表2.1 ソースステートメントで記述できる文字セット

文字	名　称	ASCIIコード
0～9	数字	0x30～0x39
A～Z	英大文字	0x41～0x5A
a～z	英小文字	0x61～0x7A
"	ダブルクォーテーション	0x22
$	ドルマーク	0x24
%	パーセント	0x25
&	アンパサンド	0x26
'	シングルクォーテーション	0x27
(	左括弧	0x28
)	右括弧	0x29
*	アスタリスク	0x2a
+	プラス	0x2b
,	コンマ	0x2c
-	マイナス	0x2d
/	スラッシュ	0x2f
:	コロン	0x3a
;	セミコロン	0x3b
<	不等号	0x3c
=	等号	0x3d
>	不等号	0x3e
¥	円マーク	0x5c
^	ハットマーク	0x5e
_	アンダースコア	0x5f
|	縦線	0x7c
~	波線	0x7e
(SP)	空白(スペース)	0x20
(TAB)	空白(タブ)	0x09

コメントの中では表 2.1 以外の文字や漢字も使うことができます。また、これらの文字以外に '?','#'が、それぞれ"TUMPL", "TUAPP"で使用されます。これらを使う場合は、注意して下さい。

2.2 識別子

識別子は文字の並びで、アセンブラに対して特別な意味を持ちます。たとえば、機械命令のニモニックやユーザがある番地に定義したシンボル等です。

2.2.1 識別子の記述方法

記述規則　識別子は次の規則に従って記述します。

- 識別子の記述には、以下に示す文字が使用できます。

  英大文字　A～Z

  英小文字　a～z

  数字　0～9

  特殊記号　_(Under line),.(Period)

- 識別子は、英字または特殊記号で始まる英数字、特殊記号列です。ただし、特殊記号で始まる識別子および特殊記号で終了する識別子は、アセンブラシステムが使用する(C900から渡される)ものです。ユーザ定義シンボルとして使用しないことをお奨めします。

  1990SYM ..... 数字で始まっているのでシンボルとは見なしません

  SYM_1990 ..... シンボルとして使用できます

  _label_1 ..... シンボルとして使用できますが、推奨しません

- 識別子は32文字目までが識別されます。32文字を越えるシンボルが記述されてもエラーは出力されず、33文字目以降は無視されます。このため、以下の2つのシンボルは同じものIT_IS_A_SYMBOL_OF_MAIN_MODULE_IN と見なされます。

  IT_IS_A_SYMBOL_OF_MAIN_MODULE_IN_ASM

  IT_IS_A_SYMBOL_OF_MAIN_MODULE_IN_C

- シンボル中の英字は、予約語を除き、大文字, 小文字が区別されます。予約語は大文字、小文字が混在していても、予約語と見なされます。

  SYMBOL　symbol　異なったユーザ定義シンボルと見なされます。

  MODULE　module　同じ擬似命令と見なされます。

2.2.2 識別子の種類

識別子は、以下に示す種類に分類されます。

識別子の種類		説　明
予約語		命令ニモニック、レジスタ名等
既定義シンボル		現バージョンではサポートせず
ユーザ定義シンボル	変数名	データ領域の名前
	ラベル	プログラム領域のラベル
	擬似命令のシンボル	セクション名、EQU名など

予約語

予約語は、アセンブラにおいてあらかじめ定義されている識別子です。これには、機械命令ニモニック、擬似命令、レジスタ･フラグ名などが含まれます。

予約語一覧　擬似命令

```
ABS       ALIGN     BANK(*)   BANKOF(*) CODE      COM0(*)
COM1(*)   DATA      DB        DC(*)     DD        DFB
DFD       DFL       DFP       DFW       DL        DP
DW        DSB       DSD       DSL       DSP       DSW
END       EQU       EXPAOF(*) EXTERN    FDATAD    FDATAS
FDATAX(%) LARGE     LOC0(*)   LOC1(*)   MEDIUM    MODULE
ORG       PUBLIC    ROMDATA   SECTION   SIZEOF    SMALL
STARTOF
```

(*) はASM90専用の予約語です

制御命令

```
EJECT     INCLUDE    LIST       MAXIMUM(&)    MINIMUM(&)
NOLIST    NORMAL(&)  RESTORE    SAVE          SYSTEM(&)
TITLE
```

(&) はASM900専用の予約語です

Ｃ言語対応擬似命令

```
FILE   LINE   SYMBOL_A   SYMBOL_B   SYMBOL_N   SYMBOL_T
```

演算子

```
BANKOF(*)          FIMDH     FIMDL     FIMS      FIMXH
FIMXL     FIMXM    EXPAOF(*)           SIZEOF    STARTOF
```

(*) はASM90専用の予約語です

これら擬似命令、制御命令以外に機械命令ニモニック、レジスタ名、フラグ名などが予約語になっていますが、各アセンブラパッケージで異なりますので各アセンブラの「取扱説明書[活用編]」を参照して下さい。

既定義シンボル

アセンブラシステムがあらかじめ準備している名前であり、通常 I/O に割り当てられています。このシンボルは、改めて定義することなく、通常のユーザ定義シンボルのように使用できます。これは各アセンブラパッケージによって異なりますので、各アセンブラの「取扱説明書[活用編]」もしくは第 2 部, 2.3 節を参照して下さい。

変数名

データを取り扱うために定義する名前です。データ定義擬似命令で定義し、絶対アドレッシングモードなどで参照します。初期値のある変数名は romdata 領域で db,dfb 擬似命令などを使用して定義し、初期値のないものは、data 領域で dsb 擬似命令などを使用して定義します。以下に示したソースステートメント中で、"dataarea","sum1"が変数名です。

変数名の例

```
dataarea    dw   0xffc0         ; メモリ領域と同時にその位置を指す
        :                       ; 変数名が定義される
sum1        dsl  1
        :
label_a:    ld   a,(dataarea) ; 変数名の参照  a ←0xffc0
            ld   ix,sum1      ; 変数名の参照
                              ; ix ←Address of sum1
```

ラベル

ラベルは機械命令コードやデータ領域のロケーションを表し、ジャンプ命令やコール命令によって参照されます。ラベルは、行の最初のフィールドに、ラベルであることを明示するための":"(コロン)を付加して定義します。ラベルの定義は 1 行に 1 つだけ許され、複数個記述するとエラーとなります。一つのロケーションに複数のラベルを定義したい場合は、複数行に渡って記述してください。
ラベルの参照時は、機械命令ステートメントのオペランド領域や擬似命令ステートメントの擬似命令情報領域に、":"を付加せずに記述します。以下の下線部はラベルです。

ラベルの例

```
start:                  ; ラベルの定義
label: add  a,0x10      ; startとlabelは同位置につけられたラベル
       jr   label       ; ラベルの参照
       :
table  dw   start
```

擬似命令のシンボル

擬似命令で定義されたシンボルで、以下の種類があります。各シンボルの詳細は擬似命令の説明を参照してください。

モジュール名　module擬似命令で定義する、オブジェクトモジュールを識別するための名前です。1モジュール(ファイル)に一つ定義します。モジュール名はデバッグ時に使用します。

セクション名　section擬似命令で定義する、セクションに付ける名前です。セクションはリンク作業の単位ですので、リンク時にこのセクション名を使用します。セクションはいくつかの属性を持ち、この属性が一致しないセクションどうしはリンクできないなどの制約があります。

EQU名　equ擬似命令で値を定義する定数名です。

外部シンボル　public擬似命令で宣言する外部定義シンボルと、extern擬似命令で宣言する外部参照シンボルがあります。外部定義シンボルは、他のモジュールからシンボルの参照を可能にするよう宣言し、外部参照シンボルは、他のモジュールで定義したシンボルの参照を宣言します。

2.3 表現式

表現式は、いくつかの要素を組合せた式であり、BNF 記法で以下の様に表されます。表現式は、<CR><LF>、コメント開始記号、';'(カンマ)または、表現式を囲む最も外側の右括弧によって終了します。表現式中の括弧は、左右のバランスがとれていなければなりません。

また、予約語は表現式に含むことはできません。

表現式: 整数型数値定数
| 浮動小数点型数値定数
| 文字定数
| シンボル (ただし、セクション名、モジュール名は除く)
| ロケーション･カウンタ･リファレンス
| 単項演算子　表現式
| 表現式　二項演算子　表現式
|(　表現式　)
;

表現式は 32bit の 2 進数で評価され、負数の場合は 2 の補数で表現されます。32bit 以上のオーバーフローは無視されます。また、命令によってはオペランドとして 8 ビット、16 ビットの値を必要としますが、この範囲を超える値が指定された場合はワーニングが出力され、有効範囲のみが取り出されます。

2.3.1 整数型数値定数

整数型数値はそれ自身で固定の値を持ち、数字で始まる英数字列で表されます。全ての整数型数値はアセンブラの内部では符号付きの倍長整数(32bits)として扱われ、それ以上の溢れは無視されます。

整数型数値定数の表記法には 2 種類あります。一方は UNIX(C 言語)記法で、他方は IEEE 第 2 記法です。特に指定のない場合、整数型数値は UNIX 記法と見なされます。IEEE 第 2 記法を使用する場合は、コマンド入力時にオプションスイッチ"-H"を指定してください。一つのソースプログラム中で、両記法の混在は許されません。

UNIX記法　2進数　：0y(ｾﾞﾛ､ﾜｲ)で始まる0または1の数字列で表します。

8進数　：0(ｾﾞﾛ)で始まる0～7の数字列で表します。

10進数　：0(ｾﾞﾛ)、0x(ｾﾞﾛ､ｴｯｸｽ)または0y(ｾﾞﾛ､ﾜｲ)以外で始まる0～9の数字列で表します。

16進数　：0x(ｾﾞﾛ､ｴｯｸｽ)で始まる0～9,A～F,a～fの英数字列で表します。

IEEE第2記法　2進数　：Bまたはbで終了する0または1の数字列で表します。

8進数　：O(ｵｰ),o,Qまたはqで終了する0～7の数字列で表します。

10進数　：Dまたはdで終了する0～9の数字列、または単に0～9の数字列で表します。

16進数　：0～9で始まりHまたはhで終了する0～9,A～F,a～fの英数字列で表します。

記法比較　両記法の比較を 表2.2 に示します。

表2.2 整数型数値定数記法比較

	UNIX記法	IEEE第2記法
2進数	0y10010011　0y1111	10010011B　1111b
8進数	072541　0312　061134237　011	72541Q　312O　61134237o 11q
10進数	100　65535　12345	100D　65535d　012345
16進数	0x7FFF　0X154498　0xab5d	7FFFH　154498h　0ab5dh

2.3.2 浮動小数点型数値定数

浮動小数点数の記述場所

浮動小数点型数値定数は、浮動小数点型数値定数をデータとして確保する擬似命令（fdatas、fdatad、fdatax）のオペランド、または浮動小数点型数値定数のビットパターンを返す関数型演算子（fims、fimdh、fimdl、fimxh、fimxm、fimxl）の引き数として記述することができます。

最終評価結果が浮動小数点型定数である表現式が、本来整数を記述しなければならない場所に記述された場合は、その評価値の小数点以下が切り捨てられ、倍長整数に変換されます。また、倍長整数に変換された値が、その記述場所の型よりも大きい場合にはワーニングとなり、上位が切り捨てられます。

例

```
ld      w, 320.3 + 2.2e2
```

下線部は、まず倍精度浮動小数点数(double)型で演算され、その結果の小数点以下が切り捨てられ倍長整数に変換されます。更に、切り捨てた結果の整数が記述場所の型（バイト型）よりも大きいので、ワーニングが出力され上位が切り捨てられます。

浮動小数点数の種類

浮動小数点型数値定数には、32 ビットのデータを扱う単精度、64 ビットのデータを扱う倍精度、80 ビットのデータを扱う拡張倍精度の 3 つの型があります。以下に各型について説明します。

単精度浮動小数点数

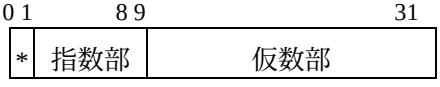
単精度浮動小数点数（float）型は IEEE 浮動小数点フォーマットに準じています。float 型は、符号部 1 ビット、指数部 8 ビット、仮数部 23 ビットの 32 ビットからなるデータです。float 型で指定可能な範囲は、

-3.40282347E+38 ～ -1.17549435E-38

+1.17549435E-38 ～ +3.40282347E+38

及び 0

であり、10進での有効桁数は7桁です。



*：符号部

倍精度浮動小数点数

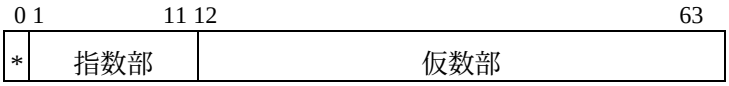
倍精度浮動小数点数（double）型は IEEE 浮動小数点フォーマットに準じています。double 型は、符号部 1 ビット、指数部 11 ビット、仮数部 52 ビットの 64 ビットからなるデータです。double 型で指定可能な範囲は、

-1.7976931348623154E+308 ～ -2.2250738585072014E-308

+2.2250738585072014E-308 ～ +1.7976931348623154E+308

及び 0

であり、10進での有効桁数は15桁です。



*：符号部

拡張倍精度浮動小数点数

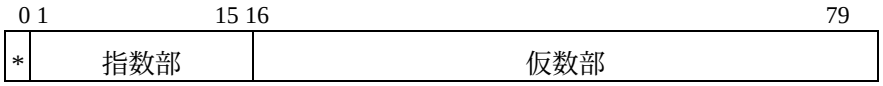
拡張倍精度浮動小数点数（long double）型は IEEE 浮動小数点フォーマットに準じています。long double 型は、符号部 1 ビット、指数部 15 ビット、仮数部 64 ビットの 80 ビットからなるデータです。long double 型で指定可能な範囲は、

-1.189731495357231499E+4932 ～ -3.362103143112093503E-4932

+3.362103143112093503E-4932 ～ +1.189731495357231499E+4932

及び 0

であり、10進での有効桁数は15桁です。



*：符号部

注　long double型の数値は、実際には倍精度浮動小数点数（double）型として処理されるため、有効桁数はdouble型と同じ15桁となります。

浮動小数点数の表記方法

浮動小数点数の表記方法は ANSI 又は ISO/IEC 9899 JIS X3010 の表記方法に準じています。但し、有効数字部がピリオドで始まるもの（.12 や .3e5F、等）についてはシンタックスエラーとなります。（上述の ANSI 及び ISO/IEC 9899 JIS X3010 では、この形式も認めています。）下記は浮動小数点数の表記文法です。

```
浮動小数点数：
        <数字列> . <数字列>  [指数部]  [浮動小数点接尾語]
    ‖  <数字列> . [指数部]  [浮動小数点接尾語]
    ‖  <数字列>  指数部  [浮動小数点接尾語]
  指数部：
        e [+‖-] <数字列>
    ‖  E [+‖-] <数字列>
  浮動小数点接尾語：
        f  ‖  l  ‖  F  ‖  L
```

注）‖: 選択　　[]: 省略可能

浮動小数点数は有効数字部を持ち、その後ろに指数部及びその型を指定する接尾語を続けて記述することができます。有効数字部の構成は、整数部を表す数字列、その後ろにピリオド（.）、その後ろに小数部を表す数字列を記述することができます。指数部の構成は、e または E の後ろに符号（省略可能）がついた数字列を記述します。有効数字部には、整数部が必要であり、かつ、ピリオドまたは指数部のいずれかが必要です。接尾語は浮動小数点数の型を表し、省略時は倍精度浮動小数点型と見なされます。接尾語と型の対応は、

fまたはF　：単精度浮動小数点型（float型）
接尾語なし　：倍精度浮動小数点型（double型）
lまたはL　：拡張倍精度浮動小数点型（long double型）

となります。

例

```
fdatas      1.2e5f    ; float型
fdatas      12e4F     ; 1行目と同じ値
fdatad      0.25e-3   ; double型
fdatad      2.5e-4    ; 3行目と同じ値
fdatax      0.2e2l    ; long double型
fdatax      20.0L     ; 5行目と同じ値
```

この例では、各型の浮動小数点数のデータ領域を確保し、その確保した領域を初期化しています。（擬似命令 fdatas、fdatad、fdatax については、第 4 章「アセンブラの擬似命令」で詳説します。）

2.3.3 文字定数

文字定数は、シングルクォーテーション(')で囲まれた 1 文字以上の印字可能な記号(ASCII 文字)で表現されます。文字定数は、それ自身で固定の 32 ビット(4 バイト)の値を持ちます。ASCII コードは、記述された順に上位からセットされます。文字が 4 バイトに満たない場合は、上位が 00 で埋められ、下位に ASCII コードがセットされます。

```
; 'A'=0x00000041   'ab'=0x00006162
charac      db   'A'       ; db      0x41
integ       dw   'ab'+3    ; dw      0x6165
label:      ld   a,'0'     ; ld      a,0x30
```

エスケープ文字

文字定数には、2種類のエスケープ文字が使用できます。一つは、'¥'の後に数値を直接記述するエスケープ文字です。'¥'の後に値の範囲が0～377の3桁以内の8進数を記述するか、もしくは'¥x'の後に0～ffの2桁の16進数を記述します。
もう一方は、次に示す特殊なエスケープ文字です。

¥0 ... 0x00 (Null)	¥a ... 0x07 (Alert , Bell)
¥b ... 0x08 (Backspace)	¥f ... 0x0c (Formfeed)
¥n ... 0x0a (Line feed)	¥r ... 0x0d (Carriage return)
¥t ... 0x09 (Horizontal tab)	¥v ... 0x0b (Vertical tab)
¥" ... 0x22 (")	¥' ... 0x27 (')
¥? ... 0x3f (?)	¥¥ ... 0x5c (¥)

2.3.4 ロケーション･カウンタ･リファレンス

オペランド中の記号'$'は、現在のセグメントのロケーション･カウンタ値の参照を意味します。現ロケーションカウンタ値とは、次の意味で使用されます。

- オペランドに'$'を持つ命令がメモリを占有する命令の時は、その命令によって占められるメモリの先頭アドレスとなります。
- メモリを占有しない命令の時は、直前までの命令によって占められた最後のメモリの次のアドレスとなります。ただし、先行する命令がORG命令の場合はORGによる更新後のロケーションカウンタ値です。

```
        LD   IX,$      ; IX <- Current location counter
LOOP:   JP   $         ; Self Jump
                       ; Same as 'LOOP: JP LOOP'
        ORG  $+0x100   ; Location counter + 0x100
```

2.3.5 式

式は、機械命令や擬似命令のオペランドを指定するときに使用します。式は、1 つまたは、複数の項と演算子の組み合わせで構成されます。

式の分類　式は、値の確定時期による分類ができます。式の値がアセンブル時に確定するか、またはリンク処理の後に確定するかで、以下の2種類に分類されます。

絶対式　式の値がアセンブル時に確定する式です。この値（アブソリュート値）はリンク処理で変更されません。

相対式　相対値、外部参照シンボルなどの、アセンブル時に値が確定せずリンク処理後に値が確定する項を含む式を、相対式といいます。また、相対式はさらに単純相対式と複合相対式に分類されます。

- 単純相対式
  - (a) 正の相対値を1つだけ含む式
- 複合相対式
  - (a) 負の相対値を含む式
  - (b) 2つ以上の相対値を含む式
  - (c) 外部参照シンボルを含む式
  - (d) セクション名を含む式
  - (e) 上記の4つの式に加減算を組み合わせた式

同一セクション内の相対値間の減算の結果は絶対値になります。

式の値　式の値は文字列を除いて符号付きの32ビット整数として計算されます。計算中にオーバーフローが発生した場合には、32ビットより上の部分は無視されます。
式の値の範囲とサイズとの関係を表 2.3 に示します。

表2.3: 式の値の範囲

サイズ	符号付き整数	符号無し整数
バイト	-128～127	-256～255
ワード	-32768～32767	-65536～65535
ロング	-2147483648～2147483647	-4294967296～4294967295

以下では、上記の分類に従ってそれぞれの式を説明し、その後に演算子について説明します。

2.3.6 絶対式(定数式)

表現式の中で、評価後にアブソリュート値(確定した値)が算出される式を特に絶対式と呼びます。評価後に値が確定していなければならないことから、外部参照シンボル、リロケータブルシンボル、評価時未定義のシンボルが含まれる式は絶対式とはなりません。また、表現式と同様、予約語は絶対式には含まれません。

絶対式: 数値定数
| 文字定数
| 既定義のアブソリュートセクション内で定義されたシンボル
| アブソリュートセクション内のロケーションカウンタ
| 単項演算子　絶対式
| 絶対式　二項演算子　絶対式
|(　絶対式　)
;

```
BUF_SIZE    equ       256
dsegment    section   data medium
buffer      dsb       BUF_SIZE*2      ; dsb 512 Absolute
                                      ; expression
            dsw       buffer          ; !! Error !! Not
                                      ; absolute expression
```

2.3.7 演算子

演算子の種類

式で指定できる演算子には、算術演算子、論理演算子、関数型演算子があります。表 2.4 に各演算子の種類を示します。これらの演算子の中で浮動小数点型定数を扱えるのは、算術演算の中の四則演算と、関数型演算子 fims、fimdh、fimdl、fimxh、fimxm、fimxl のみです。浮動小数点型定数を含む演算の規則については後述します。また、各関数型演算子の機能についても後述します。

表2.4: 演算子一覧（次頁に続く）

算術演算子

演算子	記号	表記
単項マイナス	－	-項
加算	＋	項+項
減算	－	項-項
乗算	＊	項*項
除算(%)	／	項/項
剰余算	％	項%項
右シフト(&)	＞＞	項>>項
左シフト(&)	＜＜	項<<項

(&)右シフト、左シフトの2項目は絶対値でなければなりません。

(%)除算演算で除数が0の場合は、エラーとなります。

論理演算子

演算子	記号	表記
単項否定	~	~項
論理積	＆	項&項
論理和	｜	項|項
排他的論理和	＾	項^項

※関数型演算子一覧は次頁に示します。

表2.4: 演算子一覧（前頁の続き）

関数型演算子

演算子	記号	表記
セクションサイズ	sizeof	sizeof(セクション名)
セクションスタートアドレス	startof	startof(セクション)
Bank番号 (*)	bankof	bankof(ﾗﾍﾞﾙor変数名)
シンボルの下位16ビット (*)	inbank	inbank(ﾗﾍﾞﾙor変数名)
Expa値 (*)	expaof	expaof(ﾗﾍﾞﾙor変数名)
単精度浮動小数点数のビットパターンを返す	fims	fims(表現式)
倍精度浮動小数点数の上位32ビットのビットパターンを返す	fimdh	fimdh(表現式)
倍精度浮動小数点数の下位32ビットのビットパターンを返す	fimdl	fimdl(表現式)
拡張倍精度浮動小数点数の上位16ビットのビットパターンを返す	fimxh	fimxh(表現式)
拡張倍精度浮動小数点数の中位32ビットのビットパターンを返す	fimxm	fimxm(表現式)
拡張倍精度浮動小数点数の下位32ビットのビットパターンを返す	fimxl	fimxl(表現式)

(*)はASM90専用の演算子です。

演算子の優先順位

演算子には優先順位があるので、使用するときには注意が必要です。表 2.5 に演算子の優先順位を示します。

表2.5: 演算子の優先順位

優先順位	演算子
1	単項演算子(-,~)、関数型演算子(sizeof,startof等)
2	乗算、除算、剰余算(*,/,%)
3	加算、減算(+,-)
4	シフト演算(>>,<<)
5	論理積(&)
6	排他的論理和(^)
7	論理和(|)

浮動小数点定数を含む算術演算の規則

表 2.4 の算術演算子の中で、浮動小数点型定数を扱えるのは、四則演算（加減乗除）のみです。算術演算は倍精度浮動小数点型（double 型）で行われます。以下にいくつかの例を示します。

- ❑ 浮動小数点型定数どうしのみの演算

  例　fdatad 1.2e5 + 3.33 * 2.5e-3

  下線部は倍精度浮動小数点型（double型）で演算されます。

- ❑ 浮動小数点型定数と整数の演算

-- 演算子の優先順位１番が浮動小数点型定数どうしの演算の場合 --

  例　fdatad 1.2e5 + 3.33 * 2.5e-3 + 10

  下線部は倍精度浮動小数点型（double 型）で演算されます。

-- 演算子の優先順位１番が整数どうしの演算の場合 --

  例　fdatad 1.2e5 + 3 * 10

  3*10が倍長整数型で演算された後、その結果と1.2e5が倍精度浮動小数点型（double型）で演算されます。

- ❑ 型が異なる項どうしの演算結果の型

-- 演算結果の代入先の型が浮動小数点型の場合 --

  例　fdatas 10 + 1.2e5L - 2.0e-3F

  下線部の演算結果は一旦倍精度浮動小数点型（double型）となった後、fdatasなのでワーニングとなり単精度浮動小数点型（float型）に変換されます。

-- 演算結果の代入先の型が整数型の場合 --

  例　ld　　w, 320.3 + 2.2e2L

  下線部の演算結果は一旦倍精度浮動小数点型（double型）となった後、小数以下が切り捨てられ倍長整数型に変換されます。更に、倍長整数型に変換された後の値が、記述された場所の型よりも大きいので、ワーニングとなり上位が切り捨てられます。

関数型演算子

関数型演算子は、C 言語などの関数のように演算子を表すシンボルに続けて引数を括弧で囲んで指定する形式をとる演算子です。関数型演算の結果は 32 ビットとみなされます。

関数型演算子の表記について

- 関数型演算子と"("の間に空白を入れることはできません。
- 関数型演算子は大文字と小文字の表記を区別しません。

sizeof　セクションのサイズを得る

【記述形式】

```
sizeof(<セクション名>)
```

【機能】 sizeofは<セクション名>で指定したセクションのサイズを返します。

【説明】

- <セクション名>で指定したセクションのサイズが返されます。セクションのサイズは全てのファイルのリンクが終了するまでは確定しません。従って、返り値は外部参照命令externで宣言されたシンボルの値と同じ扱いになります。
- <セクション名>は、同一モジュール内に存在するsection擬似命令で定義されたセクション名を指定します。モジュール内に存在しないセクション名を指定した場合は、エラーとなります。ただし、該当する<セクション名>が本命令より後に出てきても(前方参照)問題はありません。

【例】 以下の例では、3個のセクションmain,hundler,edit_tableのサイズの合計がwaに転送されます。

```
ld      wa , sizeof(main)
add     wa , sizeof(hundler)
add     wa , sizeof(edit_table)
```

startof　セクションの開始アドレスを得る

【記述形式】

```
startof(<セクション名>)
```

【機能】 startofは<セクション名>で指定したセクションの開始アドレスを返します。

【説明】

- <セクション名>はsection擬似命令で定義したセクション名です。
- <セクション名>で指定したセクションの開始アドレスが返されます。セクションの開始アドレスは全てのファイルのリンクが終了するまでは確定しません。従って、返り値は外部参照命令externで宣言されたシンボルの値と同じ扱いになります。
- <セクション名>は、同一モジュール内に存在するsection擬似命令で定義されたセクション名を指定します。モジュール内に存在しないセクション名を指定した場合は、エラーとなります。ただし、該当する<セクション名>が本命令より後に出てきても(前方参照)問題はありません。

【例】 以下の例では、mainセクションの開始アドレスがレジスタxixに転送されます。

```
ld      xix , startof(main)
```

expaof　expa値を得る (TLCS-90に対してのみ有効)

【記述形式】

```
expaof(<ラベルまたは変数名またはセクション名>)
```

【機能】 expaofは<ラベルまたは変数名またはセクション名>で指定したローカル0あるいはローカル1セクションを写像するためのexpa0,expa1の設定値を返します。

【説明】

- <ラベルまたは変数名>は、サイズがlargeでエリア属性がloc0あるいは、loc1のセクション内で定義され、publicあるいはexternで定義されたラベルまたは変数名を指定します。
- <ラベルまたは変数名>で指定した場合は、指定したラベルまたは変数名が含まれるセクションに対するexpa0,expa1の値が返されます。
- <セクション名>は、同一モジュール内に存在するsection擬似命令で定義されたセクション名を指定します。モジュール内に存在しないセクション名を指定した場合は、エラーとなります。ただし、該当する<セクション名>が本命令より後に出てきても(前方参照)問題はありません。
- セクションの先頭アドレスは、リンク後に確定しますのでexpaofの値は、リンクが終了するまでは確定しません。

【例】 以下の例では、local0nセクションを写像するための設定値がローカル0エクスパンドレジスタexpa0に転送されます。

```
ld     (expa0) , expaof(local0n)    ; TMP90C845の場合
ldw    (expa0) , expaof(local0n)    ; TMP90C051の場合
```

bankof　bank番号を得る (TLCS-90に対してのみ有効)

【記述形式】

```
bankof(<ラベルまたは変数名またはセクション名>)
```

【機能】 bankofは<ラベルまたは変数名またはセクション名>で指定したセクションのバンク番号を返します。

【説明】

- <ラベルまたは変数名>は、publicあるいはexternで定義されたラベルまたは変数名を指定します。
- <ラベルまたは変数名>で指定した場合は、指定したラベルまたは変数名が含まれるセクションに対するバンク番号が返されます。
- <セクション名>は、同一モジュール内に存在するsection擬似命令で定義されたセクション名を指定します。モジュール内に存在しないセクション名を指定した場合は、エラーとなります。ただし、該当する<セクション名>が本命令より後に出てきても(前方参照)問題はありません。
- セクションのアドレスは、リンク後に確定しますのでbankofの値は、リンクが終了するまでは確定しません。

【例】 以下の例では、bank_0セクションのバンク番号がバンクレジスタbxに転送されます。

```
ld      (bx) , bankof(bank_0)
```

inbank　シンボルの下位16ビットを得る(TLCS-90に対してのみ有効)

【記述形式】

```
inbank(<ラベルまたは変数名>)
```

【機能】 inbankは<ラベルまたは変数名>で指定したシンボルの下位16ビットを返します。

【説明】

- <ラベルまたは変数名>は、サイズがlargeのセクション内で定義されたラベルまたは変数名を指定します。
- エリア属性loc0あるいはloc1として定義されたセクション内のラベルまたは変数名を、バンク形式でアクセスする場合に使用します。

【例】 以下の例では、エリア属性がloc0のセクション内の変数名local00をバンク形式でアクセスしています。local00の物理アドレスの下位16ビットが、レジスタixに転送されます。inbankを使用しない場合は、通常のローカル内の変数名としてlocal00の論理アドレスがixに転送されます。

```
ld   ix , inbank(local00)    ; 物理ｱﾄﾞﾚｽの下位16ﾋﾞｯﾄをixに転送
ld   ix , local00            ; 論理ｱﾄﾞﾚｽをixに転送
```

fims　単精度浮動小数点型定数のビットを得る

【記述形式】

```
fims(<表現式>)
```

【機能】 fimsは<表現式>で指定した単精度浮動小数点型(float型)定数のビットを返します。

【説明】

- <表現式>の評価結果は単精度浮動小数点型定数でなければなりません。倍精度、拡張倍精度浮動小数点型定数の場合はワーニングとなり、単精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、単精度浮動小数点型定数に変換されます。

【例】 以下の例では、単精度浮動小数点型定数1.23e+5Fのビットがレジスタxwaに転送されます。

```
ld      xwa , fims(1.23e+5F)
```

fimdh　倍精度浮動小数点型定数の上位32ビットを得る

【記述形式】

```
fimdh(<表現式>)
```

【機能】 fimdhは<表現式>で指定した倍精度浮動小数点型(double型)定数の上位32ビットを返します。

【説明】

- <表現式>の評価結果は倍精度浮動小数点型定数でなければなりません。
  単精度、拡張倍精度浮動小数点型定数の場合はワーニングとなり、倍精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、倍精度浮動小数点型定数に変換されます。

【例】 以下の例では、倍精度浮動小数点型定数134e-7の上位32ビットがレジスタxwaに転送されます。

```
ld      xwa , fimdh(134e-7)
```

fimdl　倍精度浮動小数点型定数の下位32ビットを得る

【記述形式】

```
fimdl(<表現式>)
```

【機能】 fimdlは<表現式>で指定した倍精度浮動小数点型(double型)定数の下位32ビットを返します。

【説明】

- <表現式>の評価結果は倍精度浮動小数点型定数でなければなりません。単精度、拡張倍精度浮動小数点型定数の場合はワーニングとなり、倍精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、倍精度浮動小数点型定数に変換されます。

【例】 以下の例では、倍精度浮動小数点型定数134e-7の下位32ビットがレジスタxwaに転送されます。

```
ld      xwa , fimdl(134e-7)
```

fimxh　拡張倍精度浮動小数点型定数の上位16ビットを得る

【記述形式】

```
fimxh(<表現式>)
```

【機能】 fimxhは<表現式>で指定した拡張倍精度浮動小数点型(long double型)定数の、上位16ビットを返します。

【説明】
- <表現式>の評価結果は拡張倍精度浮動小数点型定数でなければなりません。単精度、倍精度浮動小数点型定数の場合はワーニングとなり、拡張倍精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、拡張倍精度浮動小数点型定数に変換されます。

【例】 以下の例では、拡張倍精度浮動小数点型定数0.2e2Lの上位16ビットがレジスタwaに転送されます。

```
ld      wa , fimxh(0.2e2L)
```

fimxm　拡張倍精度浮動小数点型定数の中位32ビットを得る

【記述形式】

```
fimxm(<表現式>)
```

【機能】 fimxmは<表現式>で指定した拡張倍精度浮動小数点型(long double型)定数の、中位32ビット（上位17ビット目から48ビット目まで）を返します。

【説明】

- <表現式>の評価結果は拡張倍精度浮動小数点型定数でなければなりません。単精度、倍精度浮動小数点型定数の場合はワーニングとなり、拡張倍精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、拡張倍精度浮動小数点型定数に変換されます。

【例】 以下の例では、拡張倍精度浮動小数点型定数0.2e2Lの中位32ビットがレジスタxwaに転送されます。

```
ld      xwa , fimxm(0.2e2L)
```

fimxl　拡張倍精度浮動小数点型定数の下位32ビットを得る

【記述形式】

```
fimxl(<表現式>)
```

【機能】 fimxlは<表現式>で指定した拡張倍精度浮動小数点型(long double型)定数の、下位32ビットを返します。

【説明】

- <表現式>の評価結果は拡張倍精度浮動小数点型定数でなければなりません。単精度、倍精度浮動小数点型定数の場合はワーニングとなり、拡張倍精度浮動小数点型とみなされます。
- <表現式>の評価結果が浮動小数点型定数の記述規則に違反した場合（整数の場合は下記の処理をするので除きます）はエラーとなり、<表現式>の値は0として処理されます。
- <表現式>に整数を指定した場合は、ワーニングとなり、拡張倍精度浮動小数点型定数に変換されます。

【例】 以下の例では、拡張倍精度浮動小数点型定数0.2e2Lの下位32ビットがレジスタxwaに転送されます。

```
ld      xwa , fimxl(0.2e2L)
```

2.4 文字列

文字列は、ダブルクォーテーション(")で囲まれた 0 文字以上の印字可能な記号(ASCII 文字)で表現されます。文字列は表現式中には記述できません。また、機械命令のオペランドともなりません。"db"擬似命令で文字列単独で使用します。
文字列には文字定数と同様のエスケープ文字が使用できます。使用可能なエスケープ文字は、「2.3.3 文字定数」を参照してください。文字列中にダブルクォーテーション自身を記述する場合には、このエスケープ文字を使用して下さい。

```
string1 db  "Message for TLCS-900 system¥r¥n"
string2 db  "abc¥"defghi¥""    ; abc"defghi"という文字列になる
label:  ld  XWA,"ABC"          ; !! Error !!
```

第3章 機械命令行

3.1 機械命令行

機械命令行はマイコンが動作するための実際の命令を記述する行です。機械命令行は 4 つの部分に分けられますが、この章では機械命令本体であるオペレーション部とオペランド部について一般的な記述規則を説明します。各 CPU に対応した機械命令の詳細は、「データブック」を参照して下さい。

機械命令行

[<ラベル>]　<オペレーション>　<オペランド>　[<コメント>]

3.2 機械命令行のオペレーションフィールド

ここでは、オペレーションフィールドについて説明します。オペランドフィールドについては、次節で説明します。また、ラベルフィールドとコメントフィールドについては、"1.1.2 ソースステートメントの構造"を参照してください。
機械命令行のオペレーションフィールドの記述形式は以下の通りです。

【記述形式】

<命令ニモニック>[<オペレーションサイズ>]

<命令ニモニック>は機械命令の名前で、"ld", "add", "nop"等です。命令ニモニックは CPU によって異なりますので、各データブックを参照して下さい。<オペレーションサイズ>は演算、転送等の処理を行うオペレーションのサイズで、"b", "w", "l", "d"等です。オペレーションサイズの指定については、指定できる CPU と指定できない CPU があります。

TLCS-870,TLCS-870/X, TLCS-90

例えばTLCS-870、TLCS-870/X、TLCS-90等では、サイズ指定を必要とする機械命令がありませんので、機械命令一覧表に記述されているニモニックのみが記述できます。

TLCS-900

TLCS-900では、通常はオペランドの型によってオペレーションの型が決まります。しかし、命令が実行されるときに扱うデータの型によってニモニックに"b","w","l"を明示的に付加しなければならない場合があります。このとき、オペレーションの型とオペランドの型が矛盾しているとエラーとなります。

```
ldw   wa,0x1000        ; ld wa,0x1000と同じ
ldw   (memory_area),0  ; オペランドからは型が判断できない
                       ; ldのみだとByte型のld命令になってしまう
ldw   xwa,0x1000       ; ニモニックとオペランドの型が違うので
                       ; error
```

3.3 機械命令行のオペランドフィールド

機械命令行のオペランドフィールドの記述形式は以下の通りです。

【記述形式】

```
<アドレッシングモード>[<サイズ>]
```

<アドレッシングモード>は各 CPU に対応するアドレッシングモードの指定で、<サイズ>はオペランドのサイズ指定です。

3.3.1 アドレッシングモード

オペランドのアドレッシングモードは、各 CPU によって異なります。詳細は各 CPU の「データブック」を参照して下さい。参考までに代表的なアドレッシングモードの表記法を以下に挙げておきます。

アドレッシングモード	表記法
レジスタ	レジスタ名
イミィディエイト	定数式
レジスタ間接	(レジスタ名)
インデックス	(レジスタ名+ディスプレースメント)
直接	(ラベルシンボル名)
絶対	シンボル名
相対	$+ディスプレースメント

3.4 リロケータブルシンボル

命令のオペランドにリロケータブルシンボル及びリロケータブルシンボルを含む表現式が記述できます。オペランドにリロケータブルシンボルを記述した場合には、リスティングファイルにリロケータブルであることを示す"R"が表示されます。また、リロケータブルオブジェクトファイルにリロケーション情報を出力し、リンケージエディタでアドレスを解決します。

CPU によっては同じ記述の機械命令に対し、複数のオブジェクトコードが存在する場合があります。たとえば次の例で、mem_adr を何ビットのアドレスとして扱うかでオブジェクトの長さが異なります。

```
ld  (mem_adr),0xff  ; mem_adrの扱いによってオブジェクトが変わる
```

mem_adr が値のわかっているアブソリュート(絶対)値であれば、その値に合わせてオブジェクトを出力します。しかし mem_adr がリロケータブルシンボルの場合は、アセンブラは mem_adr の属性を見てオブジェクトを決定します。たとえば mem_adr が small 属性を持っていればアドレスを示すオブジェクト部は 1 バイトになりますが、large 属性だと 3 または 4 バイトになります。

実際のオブジェクトの種類と長さは CPU によって異なりますので、「アセンブラ取扱説明書[活用編]」を参照して下さい。

3.5 最適化

分岐命令(JP,CALL 等)のオペランドも、その値が 8 ビット,16 ビットに収まるか否かで、命令コードとその長さが決定されます。また、通常プログラムするときは飛び先に応じて jp/jr,call/calr などと絶対分岐と相対分岐に書き分けます。この手間を省くために、機械命令表にはない最適化分岐命令が準備されています。"j","cal"と記述することによりアセンブラはいちばん短い命令コードを自動生成します。

```
lab_1: j label_2 ; label_2の位置により最適な(短い)命令ｺｰﾄﾞを生成
       cal sub_1  ; sub_1の位置により最適な(短い)命令ｺｰﾄﾞを生成
```

3.6 数値の範囲

オペランドに記述される定数式の結果(数値)の範囲は命令やアドレッシングモードによって決まります。記述された数値が期待される範囲を超えている場合は Warning が出力され、上位のあふれた部分が切り捨てられます。

第4章 アセンブラの擬似命令

4.1 アセンブラの擬似命令

アセンブラの擬似命令には、アセンブラの動作の制御などを行うためのさまざまな命令があります。アセンブラの擬似命令は、以下の機能に分類することができます。

- モジュールとセクションを制御する命令

  module　end　section　align　org

- データ領域の定義命令

  db　dw　dp　dl　dd
  dfb　dfw　dfp　dfl　dfd
  dsb　dsw　dsp　dsl　dsd
  fdatas　fdatad　fdatax
  dc

- モジュール間参照命令

  public　extern

- シンボル定義命令

  equ

命令の表記について　アセンブラの制御命令は、通常、大文字表記と小文字表記を区別しません。

以下の節では、アセンブラの擬似命令を上記の分類にそって説明していきます。

4.2 モジュールとセクションの制御

通常、1 つのプログラムを作成する際には、機能ごとにモジュール分割してソースファイルを作成します。モジュールはプログラムを構成する単位で、1 つのモジュールがそれぞれ 1 つのソースファイルに対応します。また、モジュールは 1 つ以上のセクションで構成されています。これらの関係は図 1.1 に示しました。アセンブラは、このモジュールを単位としてソースプログラムをアセンブルし、オブジェクトファイルを作成します。
モジュールとセクションの概念については「第１章 ソースプログラム」を参照して下さい。

4.2.1 モジュールの定義

各ソースファイルでモジュールを定義するには、モジュール名を宣言します。

module　モジュールの宣言

【記述形式】

```
module <モジュール名>
```

【機能】　モジュールの名前を宣言します。

【説明】
- ❑ <モジュール名>で指定したシンボルが、モジュール名となります。
- ❑ 本命令は、ソースファイルの先頭で記述して下さい。
- ❑ 本命令は、1つのモジュール内で1回しか指定することができません。1つのモジュール内で複数回指定すると、ワーニングとなり、最初の指定が有効となります。
- ❑ パラメータ指定部、基本制御命令部は本命令より先に記述しなければなりません。また、空白行、コメント行はモジュール擬似命令の前に記述可能です。
- ❑ 本命令は、省略することができます。本命令を省略した場合、ソースファイルのファイル名の拡張子を除いた部分が、モジュール名となります。

【例】

```
module  main_asm    ; main_asmという名前のモジュールを宣言
        :
       end          ; モジュールの終了
```

end　モジュールの終了

【記述形式】

```
end
```

【機能】　モジュールの終了を宣言します。

【説明】

- 本命令は、モジュールの終了を示すもので、ソースプログラムの最後に必ず指定しなければなりません。本命令を指定しない場合は、エラーとなります。
- 本命令以降にソースステートメントが存在する場合は、ワーニングとなり、本命令以降のソースステートメントは、全て無視されアセンブルされません。ただし、リスティングファイルには出力されます。

【例】

```
module  main_asm    ; main_asmという名前のモジュールを宣言
        :
       end          ; モジュールの終了
```

4.2.2 セクションの定義

セクションは、プログラムを構成する最小の論理的単位で、アセンブラによって生成されるオブジェクトコードは、全てどれかのセクションに属することになります。また、セクションはリンク処理の際にも重要な処理単位となります。
それぞれのセクションは、ソースファイルにおいて section 擬似命令で始まり、次の section 擬似命令または end 擬似命令が現われた時点で終了します。

section　セクション定義命令

【記述形式】

```
<セクション名>　section　<種類>　[<配置>]　[<アライメント>]
```

【機能】　以降に続くプログラムがどのようなセクションに属するかを宣言します。

【説明】

- <セクション名>で指定したシンボルがセクションの名前となります。
- <種類>はセクションの種類です。セクションの種類には以下のものがあります。

  code , data , romdata

- <配置>はセクションの配置属性です。配置属性には以下の2種類の指定方法があります。<配置>を省略した場合は、相対番地セクションとなり、サイズはターゲットCPUが持っているサイズの最大サイズを指定したものとみなされます。

  (1)　abs=<配置アドレス>　　絶対番地(アブソリュート)セクションの指定
  (2)　[<サイズ>]　　相対番地(リロケータブル)セクションでの指定

- <アライメント>は、セクション内でのアライメント(ロケーションカウンタの調整値)を指定します。アライメントには、セクションそのものの配置アライメントとセクション内の各命令のアライメントがあります。

以下に、各項目について詳細を説明します。

セクション名　section擬似命令では、それぞれのセクションを区別するために<セクション名>を必ず指定しなければなりません。

絶対番地セクションには同じ名前をつけてはいけません。2 つ以上の異なる絶対番地セクションに同じ名前をつけた場合、最初の絶対番地セクションのアドレスのみ有効となり、後のセクションは 最初のセクションに順次連結されます。(ワーニングエラーも出力されます)

相対番地セクションは、1 つのソースファイルの中で同じ名前のセクションを複数回定義することができます。同じ名前のセクションは、同一のセクションとみなされます。同一ファイル内の同じ名前のセクションは、アセンブルの際に結合されて、1 つのセクションにまとめられ、ロケーションカウンタも引き継がれます。そのため、同じ名前のセクションは、<種類>,<配置>,アライメントが同じでなければなりません。<種類>,<配置>,アライメントが異なるセクションを同じ名前で定義した場合は、ワーニングエラーとなり、1 番はじめに指定されたものが、有効となります。また、相対番地セクションは、リンクされるすべてのファイルを通じてのセクション名の重複も可能です。この場合も同一ファイル内の同一名のセクションと同じ様に、<種類>,<配置>,アライメントが同じでなければなりません。

セクションの種類　セクションの<種類>には、code, data, romdataの3種類があります。

セクションの<種類>では、セクションの実行可能/不可能を指定します。以下では、それぞれのセクションの種類について説明します。

code　codeセクションは、機械命令を配置する領域です。codeセクションは、実行可能な属性を持ちます。

data　dataセクションは、プログラム実行時に値を変更するデータを配置する領域です。dataセクションは、実行不可能な属性を持ちます。

romdata　romdataセクションは、プログラム実行時に値を変更しない定数を配置する領域です。romdataセクションは、実行不可能な属性を持ちます。

【例】

```
code_sec section code medium align=2
 ; mediumｻｲｽﾞ(64K以内)のcodeｾｸｼｮﾝ
        :                   ; セクション名=code_sec
        :                   ; セクショントップのアライメント=2
data_sec section data abs=0x300 ; 絶対番地dataセクション
        :                   ; 0x300番地に配置
        :                   ; 次のsectionまたはendまで有効
       end
```

codeセクション内にds命令を記述すると、ワーニングエラーとなりますが、正常な動作をします。同じようにdataセクション内に機械命令、d,df命令を記述した場合、romdataセクション内に機械命令、ds命令を記述した場合もワーニングになります。いずれの場合も、出力されるオブジェクトファイルは正常です。

セクションの配置　セクションの<配置>の指定には、以下の2種類の指定方法があります。

(1) abs=<配置アドレス>　絶対番地(アブソリュート)セクションの指定

(2) [<サイズ>]　相対番地(リロケータブル)セクションでの指定

絶対番地指定　abs文を指定したセクションは、絶対番地セクションとなります。絶対番地セクションはアセンブルの際にセクションに含まれるシンボルに対して絶対アドレスが割り当てられます。

<配置アドレス>で指定する値は、定数式でなければなりません。それ以外の場合は、ワーニングとなり、0 を指定したものとみなされます。

【例】

```
rom_code  section code abs=0 ; 絶対番地0に配置されるcode
                             ; セクション
      :
rom_data section romdata abs=0x200
                             ; 絶対番地0x200に配置される
                             ; romdataセクション
      :
```

相対番地指定　一方、<サイズ>を指定したセクション、または<配置>を指定しなかったセクションは、相対番地(リロケータブル)セクションとなります。相対番地セクションは、アセンブルの際にセクションに含まれるシンボルに対して、セクションの先頭からの相対アドレスが割り当てられ、リンク後に最終アドレスが決定します。

サイズ　<サイズ>はセクションの大きさ(最大アドレスのビット数)を示します。<サイズ>には次の3種類があります。

small　medium　large

各々の<サイズ>が示すビット数、指定可能な<サイズ>はターゲット CPU によって異なります。次の表 4.1 を参照して下さい。

表4.1: ターゲットCPUとサイズ

(単位：ビット)

種類	サイズ	TLCS-900	TLCS-90	TLCS-870/X	TLCS-870	TLCS-870/C
code	省略	24	16	24	16	16
	small	-	-	-	-	-
	medium	16	16	16	16	16
	large	24	-	24	-	-
data	省略	24	16	24	16	16
	small	8	8	8	8	8
	medium	16	16	16	16	16
	large	24	-	24	-	-
romdata	省略	24	16	24	16	16
	small	-	-	-	-	-
	medium	16	16	16	16	16
	large	24	-	24	-	-

-: 指定不可

<サイズ>省略時は、ターゲット CPU のそのセクションの種類で指定できる最大サイズ(large または medium)を指定したものと見なされます。

例えば、TLCS-900 の medium セクションは、リンク後に最終的に配置されるアドレスが、0～0xffff のセクションです。よって、このセクション内のラベル、変数名は 16 ビット以内の値としてアセンブルされます。もしサイズが"large"だったり、サイズが省略されている場合は、同じセクションでも、ラベルや変数名が 32 ビット以内の値としてアセンブルされるので、オブジェクトコードが長くなってしまいます。

【例】

```
code1   section code medium  ; ｻｲｽﾞ指定"medium",相対番地ｾｸｼｮﾝ
start:  ld      ix,0xff00    ; startは16ﾋﾞｯﾄ値を持つｼﾝﾎﾞﾙとなる
        :
code2   section code    ; 900,870xの場合はlargeの相対番地
                                ; ｾｸｼｮﾝ
        :                       ; 870,90の場合はmediumの相対番地
                                ; ｾｸｼｮﾝ
```

アライメント　セクション配置時のアライメントを設定します。

【記述形式】

```
align=[<式1>][,<式2>]
```

【説明】

- <式1>はセクションの先頭のアライメントを表し、リンケージエディタによって他のセクションと結合される際に、セクションの先頭アドレスが<式1>の倍数のアドレスに調整されます。
- <式2>は、そのセクション内のロケーションカウンタが変化する度、すなわち各命令毎に、ロケーションカウンタを<式2>で指定された数値の倍数に調整します。
- どちらの式も省略できますが、両方を同時に省略することはできません。
- <式1>は2^nでなければなりません。2^nでない場合には、ワーニングとなり、いちばん近い2^nの値まで切り上げられます。
- <式1>,<式2>で指定する値は、1以上で2^{32}を超えない数値でなければなりません。
- <式1>,<式2>が1～2^{31}の範囲外の場合は、ワーニングとなり、1を指定したものとみなされます。
- <式1>,<式2>は絶対式でなければなりません。絶対式以外を指定した場合は、ワーニングとなり、1を指定したものとみなされます。
- なお、align文を省略した場合は、"align=1,1"を指定したものとみなされます。
- 絶対番地セクションに対してアライメントを指定したときは、<式2>のみ有効となります。配置アドレスは既に決まっているため、<式1>は無視されます。

4.2.3 ロケーションカウンタの調整

ロケーションカウンタ

アセンブラは、アセンブル中のソースステートメントから生成するオブジェクトコードの先頭アドレスを保持するカウンタを持っています。このカウンタをロケーションカウンタといいます。

ロケーションカウンタは、セクションの先頭で初期化され、ソースステートメントからオブジェクトコードが生成される度にそのバイト数が加算されるため、常に次のソースステートメントから生成されるオブジェクトコードの先頭アドレスを保持しています。

ロケーションカウンタの初期化は、相対番地セクションと絶対番地セクションでは、以下のように異なります。

相対番地セクション　　常に0で初期化

絶対番地セクション　　section命令のabs文で指定した値で初期化

通常、ロケーションカウンタは、生成されたオブジェクトコードのバイト数に応じて増加しますが、このロケーションカウンタの値を意図的に変更することができます。ロケーションカウンタの値を変更するには、以下の 2 つの方法があります。

- アドレスの境界調整

  align命令を用いると、ロケーションカウンタの値を指定した値の倍数のアドレスに調整することができます。値の指定方法は、section命令のalignと同様です。

- 任意アドレスへの変更

  org命令を用いると、ロケーションカウンタの値を指定した値に変更することができます。

ロケーションカウンタの参照

ロケーションカウンタの値は、特殊なシンボル「$」を用いてソースプログラム中で参照することができます。

align　アドレス境界の整合

【記述形式】

```
align <式>
```

【機能】　ロケーションカウンタの値を、境界値になるまで調整します。

【説明】

- 現在有効なセクションに対するロケーションカウンタの値が<式>で指定した境界にない場合、<式>で指定した境界値になるまで調整(切上げ)されます。
- <式>には、この擬似命令出現時に確定している絶対式を指定します。指定できる値は、1から2^{31}の間です。2^{31}を超えた値を指定した場合は、ワーニングとなり、1を指定したものとみなされます。
- align擬似命令はsection擬似命令のalignに優先します。すなわち、section擬似命令で"align=<式>,<式>"を指定している場合は、本命令が現れた次のロケーションカウンタのみalign擬似命令の<式>の値で調整され、その次のロケーションカウンタ以降はsection擬似命令のalignの指定どおりに戻ります。

【例】

```
data1  section data medium align=2,2
abcde db      123       ; ﾘﾝｸ時にｾｸｼｮﾝの先頭が2の倍数になる
       align  4
fghij  dp     0x123456  ; fghijのみﾛｹｰｼｮﾝｶｳﾝﾀが4の倍数になる
       db     80        ; ﾛｹｰｼｮﾝｶｳﾝﾀが2の倍数になる
```

org　ロケーションカウンタ値の変更

【記述形式】

```
org <式>
```

【機能】　ロケーションカウンタの値を変更します。

【説明】

- ❑ 現在有効なセクションに対するロケーションカウンタの値が、<式>で指定した値に変更されます。すなわち、セクションが絶対番地セクションの場合は、<式>はアドレス値そのものになります。また、セクションが相対番地セクションの場合は、<式>はセクションの先頭からのオフセット値になります。
- ❑ <式>は、この擬似命令出現時に確定している絶対式です。
- ❑ ロケーションカウンタの値を現在のロケーションカウンタ値より小さい値にすることはできません。
- ❑ ベクタテーブルの設定等以外のorg擬似命令の使用は余り御薦めしません。align擬似命令を使用するか、セクションを分けるかする事を御薦めします。

【例】

```
main     section   romdata
vect1    dl
         org       0x1000   ; セクションの先頭から0x1000
abcde    db        "string" ; abcdeはmain+0x1000の値を持つ
```

4.3 データ領域定義命令

メモリ上にデータのための領域を確保したり、その確保した領域を初期化する場合には、次の 4 種類のデータ領域定義命令を用います。

d<サイズ>	：初期値付き整数データ定義命令(1)	db, dw, dp, dl, dd
df<サイズ>	：初期値付き整数データ定義命令(2)	dfb, dfw, dfp, dfl, dfd
ds<サイズ>	：初期値なし整数データ定義命令	dsb, dsw, dsp, dsl, dsd
fdata<サイズ>	：初期値付き浮動小数点データ定義命令	fdatas,fdatad,fdatax

4.3.1 初期値付き整数データ定義命令(1)

db,dw,dp,dd,dl　整数データ領域の確保とその初期化

【記述形式】

```
d<サイズ>　<表現式>[,<表現式>,....,<表現式>]
db　<文字列>または<表現式>[,<文字列>または<表現式>,....]
```

【機能】 <サイズ>で指定した大きさの整数データ領域が<表現式>の個数だけ確保され、確保されたそれぞれの領域は<表現式>の値で初期化されます。

【説明】

- <サイズ>を指定する1文字を記述することで、1つの整数に割り当てられる領域のサイズを指定します。

B	8ビット長の整数領域
W	16ビット長の整数領域
P	24ビット長の整数領域
L or D	32ビット長の整数領域

- <サイズ>は省略できません。
- <表現式>は確保された領域を初期化するための式です。
- <サイズ>で"B"を指定した場合に限り、<文字列>を指定することもできます。確保された領域を初期化するための文字列を指定します。
  なお、この場合領域の最後に¥0は付加されません。
- <表現式>には相対式も記述できます。
- <表現式>が数値に評価されない場合は、エラーとなります。
- <表現式>の値が<サイズ>で指定したサイズを超えた場合は、ワーニングとなり、オーバーフローした上位ビットは無視されます。
- <文字列>から生成される文字列の長さは最大511バイトです。この長さを超えるとエラーとなります。
- <文字列>が記述規則に違反した場合は、ワーニングとなり、その文字列に対する領域は確保されません。
- <表現式>や<文字列>の間は、必ずコンマ(,)で区切ります。
- 本命令は、種類がcode,romdataのセクションに記述できます。

【例】

```
integer   dw   100, 200, 300
          db   16, 0
string1   db   "Constant", "value"
```

4.3.2 初期値付き整数データ定義命令(2)

dfb,dfw,dfp,dfd,dfl　整数データ領域の複数確保とその初期化

【記述形式】

df<サイズ>	<個数式>,<表現式>
dfb	<個数式>,<文字列>

【機能】 <サイズ>で指定した大きさの連続した整数データ領域が<個数式>で指定した個数だけ確保され、確保された全ての領域が<表現式>の値で初期化されます。

【説明】

- <サイズ>を指定する1文字を記述することで、1つの整数に割り当てられる領域のサイズを指定します。

B	8ビット長の整数領域
W	16ビット長の整数領域
P	24ビット長の整数領域
L or D	32ビット長の整数領域

- <サイズ>は省略できません。
- <表現式>は確保された領域を初期化するための式です。
- <表現式>には相対式も記述できます。
- <表現式>が数値に評価されない場合は、エラーとなります。
- <表現式>の値が<サイズ>で指定したサイズを越えた場合は、ワーニングとなり、オーバーフローした上位ビットは無視されます。
- <個数式>は確保される領域の個数で、絶対式でなければなりません。絶対式でない場合は、エラーとなります。
- <個数式>が数値に評価されない場合は、エラーとなります。前方参照の絶対式も許されません。
- <サイズ>で"B"を指定した場合に限り、<文字列>の指定ができます。その場合は、<文字列>で指定された文字列を<個数式>で指定された数だけ複写して領域に割り当てます。
- <文字列>から生成される文字列の長さは最大511バイトです。この長さを越えるとエラーとなります。
- <文字列>が記述規則に違反した場合は、ワーニングとなり、その文字列に対する領域は確保されません。
- 本命令は、種類がcode,romdataのセクションに記述できます。

【例】

```
integer   dfl   20,0x4500
          dfb   16,0
string2   dfb   16,"data"
```

4.3.3 初期値なし整数データ定義命令

dsb,dsw,dsp,dsd,dsl　整数データ領域の確保(初期化なし)

【記述形式】

```
ds<サイズ>　　<個数式>
```

【機能】 <サイズ>で指定した大きさの連続した整数データ領域が、<個数式>で指定した数だけ確保されます。確保された領域は初期化されません。

【説明】

- <サイズ>を指定する1文字を記述することで、1つの整数に割り当てられる領域のサイズを指定します。

B	8ビット長の整数領域
W	16ビット長の整数領域
P	24ビット長の整数領域
L or D	32ビット長の整数領域

- <サイズ>は省略できません。
- <個数式>は確保される領域の個数で、絶対式でなければなりません。絶対式でない場合は、エラーとなります。
- <個数式>は正の整数でなければならず、0以下の値に対してはワーニングが出力され、1が指定されたものとみなされます。
- <個数式>が数値に評価されない場合は、エラーとなります。前方参照の絶対式も許されません。
- 本命令は、種類がdataのセクションに記述できます。

【例】

```
integer   dsw    20
          dsb    3
```

4.3.4 初期値付き浮動小数点データ定義命令

fdatas,fdatad,fdatax　浮動小数点データ領域の確保とその初期化

【記述形式】

```
fdata<サイズ>　<表現式>[,<表現式>,....,<表現式>]
```

【機能】 <サイズ>で指定した大きさの浮動小数点データ領域が<表現式>の個数だけ確保され、確保されたそれぞれの領域は<表現式>の値で初期化されます。

【説明】

- <サイズ>を指定する1文字を記述することで、1つの浮動小数点数に割り当てられる領域のサイズを指定します。
  - S　32ビット長の単精度浮動小数点データ領域
  - D　64ビット長の倍精度浮動小数点データ領域
  - X　80ビット長の拡張倍精度浮動小数点データ領域
- <サイズ>は省略できません。
- <表現式>は確保された領域を初期化するための式です。
- <表現式>の評価結果は浮動小数点定数でなければなりません。
- <表現式>の評価結果が浮動小数点定数の記述規則に違反した場合は(整数の場合は次項の処理となりますので除きます）、エラーとなり表現式の値は０として処理されます。
- <表現式>に整数を記述した場合は、ワーニングとなり、<サイズ>に合わせた浮動小数点定数に変換されます。
- <サイズ>と<表現式>とで浮動小数点数のサイズが異なる場合には、ワーニングとなり、<表現式>の値は<サイズ>で指定したサイズに変換されます。
- <表現式>で指定した値の指数部が<サイズ>で指定した浮動小数点データ指数部のサイズを越えた場合は、チェックできるものに関してはワーニングとなり、<サイズ>で指定可能な最大の指数に変換されます。
- 本命令は、種類がcode,romdataのセクションに記述できます。

【例】

```
float       fdatas   2.0e-3F,1.5e+2F,0F
double      fdatad   1.8e,3.33
longdouble  fdatax   1.25e5L
```

4.3.5 TLCS-90に対してのみ有効なデータ定義命令

dc　データ領域の確保とその初期化(TLCS-90に対してのみ有効)

【記述形式】

```
dc　<文字列>[,<文字列>,....,<文字列>]
```

【機能】 <文字列>の長さ分の整数データ領域が確保され、確保されたそれぞれの領域は指定した文字列で初期化されます。

【説明】

- ❑ <文字列>内の各文字は、先頭から順にASCIIコード値で、確保された領域に格納されますが文字列の、最後の文字の最上位ビットには、常に1がセットされます。
- ❑ <文字列>から生成される文字列の長さは最大511バイトです。この長さを越えるとエラーとなります。
- ❑ <文字列>が記述規則に違反した場合は、ワーニングとなり、その文字列に対する領域は確保されません。
- ❑ <文字列>の間は、必ずコンマ(,)で区切ります。

【例】

```
stting1   dc   "ASCII"            ; Object(16進): 41534349C9
          dc   "ASCII","Data"     ; Object(16進): 41534349C9
446174E1
```

4.4 モジュール間の参照

プログラムをモジュール分割して作成する場合、モジュールが全く独立していることは少なく、モジュール間でのシンボルの参照が発生します。セクション内で定義した変数名やラベルなどのシンボルは、そのままでは他のモジュールから参照することはできません。以下の 2 種類の命令でシンボルを宣言することにより、シンボルのモジュール間の参照が可能となります。

外部定義命令　public擬似命令はモジュール内で値を定義したシンボルを、他のモジュールから参照可能にします。この命令を外部定義命令といいます。また、この命令で宣言したシンボルを外部定義シンボルといいます。

外部参照命令　extern擬似命令はモジュール内では値が未定義で、他のモジュールで値が定義されているシンボルを参照可能にします。この命令を外部参照命令といいます。また、この命令で宣言したシンボルを外部参照シンボルといいます。

上記の命令で宣言したシンボルに対するモジュール間参照の解決(参照したシンボルの値のオブジェクトへの埋込みなど)は、アセンブル時には行われず、モジュール間の参照を解決するのに必要な情報がオブジェクトファイルに出力されます。この情報をシンボル情報といいます。モジュール間参照の最終的な解決は、リンケージエディタによって行われます。

public　外部定義シンボルの宣言

【記述形式】

```
public  <シンボル>[,<シンボル>,...,<シンボル>]
```

【機能】　<シンボル>を外部定義シンボルとして宣言します。

【説明】
- ❑ <シンボル>を、他のモジュールから参照可能にします。
- ❑ この命令で宣言できる<シンボル>は、モジュール内で値を定義したシンボルでなければなりません。
- ❑ 本命令は、ソースファイル中の任意の位置に記述できます。

【例】

```
  public  data0, data2, data4  ; 他のモジュールで参照するシンボル
data0   equ   0xffff          ; data0はpublicシンボル
data1   equ   0x00ff          ; data1はlocalシンボル
d_area  section data medium
data2   dsb   4               ; data2はpublicシンボル
data3   dsl   1               ; data3はlocalシンボル
r_area  section romdata large
data4   dw    1024            ; data4はpublicシンボル
data5   db    "string"        ; data5はlocalシンボル
```

extern　外部参照シンボルの宣言

【記述形式】

```
extern  [<サイズ>]  <シンボル>[,<シンボル>,...,<シンボル>]
```

【機能】 <シンボル>を外部参照シンボルとして宣言します。

【説明】

- extern命令では、他のモジュールで定義した<シンボル>の値を参照することを宣言します。
- externで宣言した<シンボル>は、モジュール内で値を定義することはできません。
- 本命令は、ソースファイル中の任意の位置に記述できます。
- <サイズ>には、シンボルが実際に定義されているセクション(他のファイルにある)のsection擬似命令で指定したサイズ(small,medium,large)を記述します。
- 絶対番地セクション内のシンボルをextern宣言する時には、section擬似命令の配置アドレスに該当するサイズをextern擬似命令の<サイズ>で指定します。
- section擬似命令のサイズと、本命令の<サイズ>が矛盾するとリンク時にエラーとなります。
- <サイズ>省略時は、ターゲットCPUの最大サイズが指定されたものと見なされます。

【例】

```
   extern     ext_l          ; ext_lはlarge属性を持つ外部参照
                             ; シンボル
   extern     medium ext_m   ; ext_mはmedium属性を持つ外部参照
                             ; シンボル
cod_sec1 section code medium
start1:   ld     ix,ext_m    ; 16ビット値ext_mをixレジスタへ
          ld     a,(ext_l)   ; large領域のext_1のデータを
                             ; aレジスタへ
      :
```

他のファイルでシンボルを定義し、public宣言しておきます。

```
   public  ext_l, ext_m      ; ext_l, ext_mの外部定義宣言
cod_sec2 section code large
ext_l:  ld   ix, 0xff00      ; ext_lの定義
        :
data_sec section romdata medium
ext_m   db   123             ; ext_mの定義
        :
```

4.5 シンボルの定義

アセンブリ言語では、ソースプログラム中で用いる定数に対してシンボルを定義することができます。

equ　定数シンボルの定義

【記述形式】

```
<シンボル>　equ　<式>
```

【機能】　モジュールやプログラム全体で使用する定数を<シンボル>に定義します。

【説明】

- <式>で指定した値が<シンボル>に設定されます。
- <式>は、絶対式でなければなりません。それ以外の場合はワーニングとなり、値は0になります。
- 本命令を用いて設定したシンボルの値を変更することはできません。
- 他の命令で定義したシンボルをequ命令で再定義するとエラーとなります。
- 本命令で定義した<シンボル>は、public命令で外部定義シンボルとして宣言することができます。

【例】

```
const1   equ  0          ; const1=0
const2   equ  10         ; const2=10
abcd     equ  const2     ; abcd=10
abcd     equ  20         ; error!  abcdは定義済み、再定義不可
public   const1,const2 ; const1,const2は他のモジュールで参照
                         ; される
```

第5章 アセンブラの制御命令

5.1 アセンブラの制御命令

アセンブラの制御命令には、アセンブラの動作の制御などを行うためのさまざまな命令があります。

アセンブラの制御命令は、以下の機能に分類することができます。

- 基本制御命令

  ソースプログラムの中で一度だけ設定する基本的な制御命令です。

  maximum/minimum (TLCS-900のみ)

- 一般制御命令

  ソースプログラム中で種々の状態を変更するための制御命令で、何度も記述できます。

  system/normal(TLCS-900のみ) , list/nolist , title , eject , save/restore

- ファイルインクルード命令

  ヘッダファイルなどをソースプログラムに挿入するための制御命令です。必要な場所に必要な数だけ記述します。

  include

以下の節では、アセンブラの制御命令を上記の分類にそって説明していきます。

命令の表記　制御命令の先頭には必ずドルマーク($)を付けます。アセンブラの制御命令では、通常、大文字表記と小文字表記が区別されません。また、$と制御命令の間はあけてはいけません。エラーとなりますので注意して下さい。

5.2 基本制御命令

基本制御命令は、ターゲット CPU のモードなど、一度設定した後変更しないものを宣言します。

5.2.1 TLCS-900の基本制御命令

MINIMUM / MAXIMUM　CPUモード選択(TLCS-900のみ)

【記述形式】

```
$MINIMUM　または　$MAXIMUM
```

【機能】　ターゲットCPUがTLCS-900の場合に、CPUのモードを選択します。

$MINIMUM　　ミニマムモード

$MAXIMUM　　マキシマムモード

【説明】

- 本命令はTLCS-900のアセンブラ(ASM900)に対してのみ有効です。
- 本命令はASM900では省略できません。
- 本命令は必ずソースプログラムの先頭に記述します。本命令より前に記述できるのは、パラメータ指定部、コメント行、空白行のみです。
- $MINIMUM

  TLCS-900をミニマムモードで使用する場合に記述します。"$MINIMUM"が記述された場合は、ASM900はミニマムモードとして動作します。従って、マキシマムモード特有のレジスタ(XWA,XBCなど)は使用できません。また、コード領域は64Kバイト以内なので、CODEセクションのLARGEは記述できません。
- $MAXIMUM

  TLCS-900をマキシマムモードで使用する場合に記述します。"$MAXIMUM"が記述された場合は、ASM900はマキシマムモードとして動作します。従って、ミニマムモード固有のレジスタは使用できません。コード領域は16Mバイトなので、セクションは自由に記述できます。

【注意】　本命令は、モードの混同という間違いが起こらないために、アセンブラを制御するだけです。本命令により実際にCPUのモードが切り替わるわけではありません。CPUのモードを切り替える場合は、該当する機械命令を使用して下さい。

5.3 一般制御命令

一般制御命令は、基本制御文と end 擬似命令の間なら、どこにでも何度でも記述可能です。

5.3.1 TLCS-900の機械命令の制御

SYSTEM / NORMAL 特権命令記述可/不可の切り替え(TLCS-900のみ)

【記述形式】

```
$SYSTEM　または　$NORMAL
```

【機能】 特権命令の記述可/不可を切り替えます。

$SYSTEM　特権命令記述可

$NORMAL　特権命令記述不可

【説明】

- 本命令はTLCS-900のアセンブラ(ASM900)に対してのみ有効です。
- ノーマルモードはTLCS-900シリーズにしかありません。TLCS-900L/Hではこの制御命令を使用しないで下さい。
- $SYSTEM
  $SYSTEM以降$NORMAL制御命令が記述されるまで、システムモードとみなします。この範囲内では特権命令が記述できます。
- $NORMAL
  $NORMAL以降$SYSTEM制御命令が記述されるまで、ノーマルモードとみなします。この範囲内では特権命令は記述できません。
- デフォルトは$SYSTEMです。何も指定しないとシステムモードと見なされます。

【例】

```
$system          ; $normalが現れるまで特権命令が記述できる
       di
       ei
       :
$normal          ; $systemが現れるまでは特権命令は記述できない
       ld  wa,bc
       :
```

【注意】 本命令は、誤って特権命令を使用しないようにという便宜上からアセンブラを制御するだけです。本命令により実際にCPUのモードが切り替わるわけではありません。CPUのモードを切り替える場合は、該当する機械命令を使用して下さい。

5.3.2 リストファイルの出力制御

アセンブラはソースファイルのアセンブル結果をリストファイルとして出力します。このリストファイルに関する制御は、アセンブラの起動時オプションで指定しますが、より細かな制御をリストファイルの出力制御命令で行うことも出来ます。

LIST / NOLIST　プログラムリストの出力制御

【記述形式】

```
$LIST　または　$NOLIST
```

【機能】　アセンブルリストファイルを出力/非出力します。

$LIST　　アセンブルリストが出力されます。

$NOLIST　アセンブルリストが出力されません。

【説明】

- 本命令は、"-l"オプションを指定していない場合は、無視されます。
- $LIST

  本命令以降NOLIST命令が指定されるまで、アセンブルリストファイルにリストを出力します。
- $NOLIST

  本命令以降LIST命令が指定されるまで、アセンブルリストファイルにリストを出力しません。ただし、NOLISTの範囲内にエラーが発生した場合は、エラー行とエラーメッセージをアセンブルリストファイルに出力します。
- デフォルトは$listです。何も指定しないとリストファイルに出力されます。

【例】

```
$nolist
    ; この部分のリストは出力されない
$list
    ; この部分のリストは出力される
$nolist
```

TITLE　リストファイルのタイトルの指定

【記述形式】

```
$TITLE  "<文字列>"
```

【機能】 リストファイルのタイトルを設定します。

【説明】

- <文字列>をタイトルとしてアセンブルリストファイルのヘッドに出力します。
- 本命令が現れた時点で改ページを行い、次のページの先頭にタイトルを出力します。
- <文字列>は60文字まで指定可能です。60文字を超えた場合は、60文字以降は無視されます。
- <文字列>は印字可能なASCII文字です。
- アセンブラの起動時オプションが"-l"を指定していない場合、または本命令が記述された場所がnolist制御命令の範囲内の場合は、本命令は無効となります。
- アセンブラの起動時オプションで"-T"を指定した場合、本命令は無効になります。
- title制御文が記述されていない場合は、$title "アセンブルリストファイル名"が指定されているものと見なされます。

【例】

```
      module  atest_1
$title "Printer Test Program /"Font/""  ; 1ﾍﾟｰｼﾞからこのtitle
   :
$title "Printer Test Program /"Color/"" ; 改ﾍﾟｰｼﾞ,新ﾀｲﾄﾙが出力
   :
```

EJECT　リストファイルの改ページの指定

【記述形式】

```
$EJECT
```

【機能】　リストファイルに改ページします。

【説明】

- 本命令の後で、改ページを行います。
- アセンブラの起動時オプションで "-lpXX" を指定していない場合、または本命令が記述された場所がnolist制御命令の範囲内の場合は、本命令は無効となります。

【例】

```
       ld  (memory_1),a
       ret
$eject                          ; ここで改ﾍﾟｰｼﾞされﾀｲﾄﾙが出力される
sect_1  section code medium     ; 新しいﾍﾟｰｼﾞに出力される
sub_2:  ld     de,hl
```

SAVE / RESTORE　LIST/NOLIST状態の記憶、再現

【記述形式】　$SAVE　または　$RESTORE

【機能】　list/nolistの状態をsaveスタックに記憶、再現します。

$save　　list/nolistの状態を記憶します。

$restore　　list/nolistの状態を再現します。

【説明】　❑　本命令は、汎用マクロプロセッサTUMPLが出力するもので、一般のユーザがソースプログラムに記述することはありません。

【例】

```
$nolist
$save     ; 今のlist/nolist状態(この場合は"nolist")を記憶
   :
$list
$restore  ; "$save"が記述された時の状態に戻る
   :
```

5.4 ファイルインクルード機能

ファイルインクルード機能とは、アセンブルの際に、ソースファイルの指定した位置に別のソースファイルを 取り込む機能です。この機能は、定数の定義やマクロの定義、あるいはシンボルの外部宣言などを複数のソースファイルで利用する場合に用います。

ファイルインクルード機能を利用するには、$include 命令を用います。

include　ファイルインクルード命令

【記述形式】

```
$include "<ファイル名>"
```

【機能】 <ファイル名>で指定したソースファイルを読み込みます。

【説明】

- <ファイル名>で指定したソースファイルを読み込み、$include命令の位置に展開してアセンブルを行います。
- $include命令は、8段階までネストすることができます。これを超えるネストを行うと、エラーとなります。
- <ファイル名>を絶対パス名付きで指定した場合は、そのファイルがそのまま本命令の位置に挿入されます。ファイルが存在しなかった場合は、エラーとなります。
- <ファイル名>をパス名なしまたは相対パス名付きで指定した場合は、ファイルが以下のディレクトリの順番で検索されます。
  (1) アセンブル中のソースファイルがあるディレクトリを基準にして検索されます。
  (2) アセンブラ起動時に、"-I"オプションで指定したディレクトリを基準にして検索されます。複数指定した場合は、指定順に検索されます。
  (3) 標準ディレクトリが検索されます。標準ディレクトリとは、以下のディレクトリです。
    - 環境変数THOMExxxを設定している場合は、$THOMExxx/includeディレクトリです。(xxxはCPU名)
    - 環境変数THOMExxxを設定していない場合は、環境変数THOME/includeディレクトリです。
    - 環境変数THOMExxxとTHOMEの両方が指定された場合は、THOMExxx, THOMEの順に検索されます。
    - いずれも設定していない場合は、カレントディレクトリの下のincludeディレクトリです。

MS-DOS 版では、環境変数 THOMExxx または THOME の設定がドライブ指定のみの場合は、そのドライブの¥include ディレクトリが検索されます。

【注意】　インクルードファイル中には、基本制御文とオプション定義は記述できません。

【例】

```
       module  sample
$include       "define.inc"
   :
```

例の説明　このソースファイルが「tlcs900.asm」という名前でworkというディレクトリに保存されていて、アセンブラを

asm900　work/tlcs900.asm

と起動したと仮定します。この場合、インクルードファイル"define.inc"は次の順番で検索されます。

1. workディレクトリ
2. 標準のディレクトリ

また上記の例をアセンブルする際アセンブラの起動時オプションでサーチパスを

-I/usr/include -I../headers

と指定した場合は、ファイルは次の順番で検索されます。

(1) workディレクトリ
(2) /usr/includeディレクトリ
(3) ../headersディレクトリ
(4) 標準のディレクトリ

第 6 章 TX19 アセンブラ固有仕様

この章では TX19 アセンブラ固有の仕様について説明します。なお、TX19 固有ではないアセンブラの仕様については、第 1 章～第 5 章を参照して下さい。

6.1 アライメント

6.1.1 アライメントのデフォルト値

第一部 導入 2.1 節にありますように、セクションの指定は次の書式に従います。

```
<セクション名> section <種類> [<配置属性>] [<アライメント>]
```

アライメントは省略可能です。アライメントを省略した場合の、デフォルトのアライメントを下表に示します。

abs 以外のセクションでアライメントを省略すると、アライメントは次の値を持ちます。

種類	isa32		isa16		large		medium		small	
	開始	内部	開始	内部	開始	内部	開始	内部	開始	内部
code	4	4	4	2	4	4	4	2	—	—
data	—	—	—	—	4	4	4	4	4	4
romdata	—	—	—	—	4	4	4	4	4	4

表の－は、使用できない種類と配置属性の組み合わせです。

6.1.2 abs セクションでのアライメント指定

abs を用いて配置(絶対番地)アドレスを指定することができます。abs を用いたセクション定義は以下のような書式を取ります。

```
<セクション名> section <種類> [abs=<配置アドレス>] <[アライメント]>
```

アライメント部分は省略可能で、アライメント部分を省略すると次と同じ意味になります。

```
<セクション名> section <種類> [abs=<配置アドレス>]  align=1,1
```

abs セクションの場合、前節 6.1.1 の abs 以外のセクションとはアライメントのデフォルト値が異なりますので、注意が必要です。

6.2 制御命令

$reorder

TX19 アセンブラでは、デフォルトで遅延スロットに nop を挿入しません。遅延スロットに nop を挿入するためには、次の$reorder 命令を使用します。

```
$reorder
```

本命令を記述した際、それ以降に出現する遅延スロットに nop が挿入されます。

$noreorder

```
$noreorder
```

$noreorder は、$reorder 命令の機能をキャンセルする機能を持ちます。$reorder と$noreorder を使用した例を示します。

```
f_code section code large

bgez   r2,0x0             ; nop 挿入なし

$reorder
bgez   r2,0x4             ; nop 挿入あり
bnel   r2,r4,0x8          ; nop 挿入あり

$noreorder
bgez   r2,0x4             ; nop 挿入なし
bnel   r2,r4,0x8          ; nop 挿入なし
jalr   r6                 ; nop 挿入なし

  end
```

$reorder 命令は、次に$noreorder 命令が出現するまで、セクションをまたがっても有効です。

6.3 演算子

sdaoff

```
sdaoff(<式>)
```

<式>で指定したアドレスから gp レジスタの内容を引いた、符号付き 32 ビット直値を返します。

<式>は<シンボル>または<直値>で構成された式となります。<シンボル>の場合、そのシンボルの置かれたアドレスを値として取り扱います。

lo

```
lo(<式>)
```

<式>で指定した値、もしくはシンボルのアドレスの符号なし下位 16 ビット直値を返します。

hi

```
hi(<式>)
```

<式>で指定した値、シンボルのアドレスの上位 16 ビットと、lo(<式>)の最上位ビットを加算した符号なし 16 ビット直値を返します。

pcoff

```
pcoff(<シンボル>)
```

<シンボル>のアドレスから、現在の pc レジスタの内容を引いた、符号付き 32 ビット直値を返します。

pcoff は命令のオペランドとしてのみ利用できます。

__gp

```
__gp
```

既定義メモリである data.m と romdata.m の先頭アドレスで、アドレスが若い方(数値の小さい方)に、0x00008000 を加算した符号付き 32 ビット直値と置換します。

演算子使用例

sdaoff と__gp を使用した例を示します。

```
f_code section code large

dw  __gp
dw  sdaoff(0x00017000)

end
```

アセンブル実行後、リンクします。メモリマップは以下のようになっているとします。

```
 memory {
       data.s         :      org=0x00000000, len=0x00004000
       romdata.s      :      org=0x00004000, len=0x00004000
       data.m         :      org=0x00010000, len=0x00008000
       romdata.m      :      org=0x00018000, len=0x00008000
       code.m         :      org=0x00008000, len=0x00008000
       code.l         :      org=0x00008000, len=0x00008000
}
```

その結果を以下に示しました。

```
00008000  00018000        SLL     r16,r1,0
00008004  FFFF8000        dw      0x00018000
```

data.m と romdata.m の先頭アドレスの数値が小さい方は data.m の先頭アドレス 0x000100000 になります。これに 0x00008000 を加えた値である 0x00018000 が、__gp の結果として出力されています。

0x00017000 からこの__gp の結果を引いた値 0xfffff000 が、sdaoff(0x00017000)の値として返されていることになります。

6.4 データ領域定義命令

データ領域を確保したり確保した領域を初期化するためのデータ定義疑似命令は、code, data, romdata のすべてのセクションで使用することができます。

サイズ	初期値付き定義	初期付きブロック定義	初期値なし定義
8 ビット長整数領域	db	dfb	dsb
16 ビット長整数領域	dh	dfh	dsh
24 ビット長整数領域	dp	dfp	dsp
32 ビット長整数領域	dw	dfw	dsw
64 ビット長整数領域	dl	dfl	dsl

dh, dfh, dsh は TX19 アセンブラで新規に追加された定義命令です。またその他の定義命令も、TX19 アセンブラと TLCS-900 シリーズなど、他の東芝オリジナルマイクロプロセッサのアセンブラとではサイズが異なりますので、注意が必要です。

6.5 浮動小数点数演算処理

単精度浮動小数点数

単精度浮動小数点数は、有効桁数 7 桁までです。

倍精度浮動小数点数

倍精度浮動小数点数は、有効桁数 15 桁までです。

さらに浮動小数点以下の有効桁数が 8 桁以内で記述する必要があります。有効桁数が 15 桁以内で表現できる値であっても、正規化前で浮動小数点以下の有効桁数が 15 桁を超えるような記述方法の場合、正しいオブジェクトの出力を保証しません。

以下に、fdatas, fdatad を用いた使用例を示します。

```
n_const section romdata medium align=4,4

fdatas   123.45670000e+0            ; 有効桁数内
fdatas   12.345670000e+1            ; 有効桁数内
fdatas   123.45678900e+0            ; 有効桁数外
fdatas   12.345678900e+1            ; 有効桁数外

fdatad  -12.345678901234000000000e+0     ; 有効桁数内
fdatad  -1.2345678901234000000000e+1     ; 有効桁数内
fdatad  -12.345678901234567800000e+0     ; 有効桁数外
fdatad  -1.2345678901234567800000e+1     ; 有効桁数外

fdatad  -0.012345678901234500000000e+3   ; 有効桁数外

  end
```

6.6 拡張命令 (16 ビット ISA モード)

16 ビット ISA 基本命令は、jal, jalx 命令を除いて通常 16 ビット長オブジェクトを出力しますが、直値の範囲が 16 ビット ISA 基本命令で扱えない範囲にある場合、アセンブラはデフォルトで自動的に拡張命令とし、32 ビット長のオブジェクトを出力します。

拡張命令と成り得る命令は、16 ビット ISA 基本命令で、直値を扱うオペランドを持つすべての命令です。

16 ビット ISA 基本命令で、明示的に拡張命令を指定する場合は、命令の先頭に "E" もしくは "e" を付加します。"E" 、もしくは "e" を 16 ビット ISA 基本命令に付加すると(可能であれば)無条件に、拡張命令に展開します。

以下に、拡張命令を使用した場合の例を示します。

```
n_code section code medium

addiu  r2,r4,0x4      ; 16 ビット基本命令処理
addiu  r4,r6,0xf      ; デフォルトで拡張命令に展開
lb     r2,0x10(r4)         ; 16 ビット基本命令処理
lb     r4,0xff(r4)         ; デフォルトで拡張命令に展開

eaddiu r2,r4,0x4      ; 拡張命令に展開
eaddiu r4,r6,0xf      ; 拡張命令に展開
elb    r2,0x10(r4)         ; 拡張命令に展開
elb    r4,0xff(r4)         ; 拡張命令に展開

  end
```

6.7 合成命令

nop 命令

nop 命令は常に合成命令に展開します。32 ビット ISA モードでの nop は以下の命令と同じ意味になります。

```
sll    r0,r0,0x0
```

16 ビット ISA モードにおける nop は次の命令と同じ意味になります。

```
move   r2,r2
```

16 ビット ISA モードにおける合成命令は、nop のみで、それ以外の合成命令は 32 ビット ISA モードで使用します。

命令展開

合成命令は、「The MIPS Programmer's Handbook」(*) 5. Instruction Set Reference にある命令をサポートしています。

32 ビット ISA 基本命令で、合成命令が存在し、扱える直値の範囲を超えた場合には、デフォルトで合成命令に展開します。

合成命令の許容範囲外の直値が指定された場合は、基本命令で許容外の直値が指定された時と同様、アセンブラは以下のメッセージを出力します。

```
ASM9I-Warning-501: Operand value is out of range
```

リンカによるシンボル解決の結果、オペランドに指定した数値が許容範囲外になった場合には、リンカが以下のメッセージを出力します。

```
TULINK-Warning-526:
Section <セクション名>at <アドレス>load value overflow. Truncated
```

(*) The MIPS Programmer's Handbook,
Erin Farquhar and Philip Bunce, Morgan Kaufmann Publishers, 1994

6.8 追加メッセージ

519 : Not support coprocessor register

コプロセッサ命令で、サポートされていないコプロセッサを指定すると、このメッセージを出力します。

ただし、MIPS アーキテクチャは、コプロセッサは最大 4 個 (CP0～CP3)しかサポートしていないため、CP4 以上を指定した場合には、200 : Syntax error を返します。

520 : R0 is always zero

レジスタ R0 は常に値が 0 であるレジスタで、デスティネーションレジスタとして記述するとこのメッセージを出力します。

このメッセージは、-zw オプションによって抑制されます。

521 : R1 is reserved register for macro

レジスタ R1 は、合成命令展開時などに一時的に値を保存するレジスタで、アセンブラソース中に記述するとこのメッセージを出力します。

このメッセージは、-zw オプションによって抑制されます。

6.9 追加オプション

-Rei

32 ビット ISA

基本命令と同じオペレーションコードを持つ合成命令に有効で、合成命令の展開を抑制します。その際、"ASM9I-Warning-501" を出力します。

基本命令と異なるオペレーションコードを持つ合成命令には無効で、合成命令への展開を行います。

16 ビット ISA

32 ビット拡張命令を持つすべての 16 ビット ISA 基本命令に有効で、32 ビットへの拡張を抑制します。その際、"ASM9I-Warning-501"を出力します。32 ビット拡張命令を持たない 16 ビット ISA 基本命令には無効です。

-Rmi

32 ビット ISA

基本命令と異なるオペレーションコードを持つ合成命令に有効で、合成命令の展開を禁止します。その際、"ASM9I-Error-200,230"を出力します。基本命令と同じオペレーションコードを持つ合成命令には無効で、合成命令展開を行います。

16 ビット ISA

16 ビット ISA 命令には無効です。

-Rbe

ビッグエンディアンとしてアセンブルを行います。

-Rle

リトルエンディアンとしてアセンブルを行います。

リトルエンディアンでオブジェクトを出力したい場合は、アセンブル実行時にこのオプションを指定しなければなりません。

-zc

アセンブル実行時、"ASM9I-Warning-501, 505, 517, 518"の出力を抑制します。

-zn

32 ビット ISA にのみ有効で、nop を除いたすべての合成命令の展開を抑制、または禁止します。

-zw

レジスタ R0, R1 に書き込みを行うとデフォルトでは、"ASM9I-Warning-520, 521"を出力しますが、これを抑制します。

第 1 章～第 5 章の内容は一部、TX19 には当てはまらないものがあります。それらは TX19 リンカ固有の仕様とする部分であり、第 6 章を参照して下さい。

第1章 リンケージエディタの概要

1.1 概要

汎用リンケージエディタ TULINK は、コンパイラやアセンブラで分割して作成された複数のリロケータブルオブジェクトファイルをリンクし、メモリ配置を行って、一つのアブソリュートオブジェクトファイルを生成します。リンクの過程で、ライブラリファイルからの必要なモジュールの取り込みも行います。
また、TULINK は、全てのアドレスが決定しているアブソリュートオブジェクトファイルを作成するだけでなく、アドレスの決定を保留して、再リンク可能なリロケータブルオブジェクトファイルを作成する、インクリメンタルリンクもサポートしています。
以下では TULINK を、単にリンケージエディタまたはリンカと呼びます。

リンカの特長　以下にリンケージエディタの特長を示します。

- リンケージエディタはコンパイラやアセンブラが出力したリロケータブルオブジェクトファイルをリンク処理し、メモリアドレスの決定したアブソリュートオブジェクトファイルを作成します。
- リンク作業の手順はすべてコマンド言語ファイルによって指示されます。
  コマンド言語ファイルでは、
  ターゲットのメモリ構成の指定
  結合するセクションと結合の順序の指定
  セクションを配置するアドレスおよび範囲の指定
  広域シンボルの定義、再定義
  など、細かな指示を行うことが可能です。
- デバッグ情報を出力オブジェクトファイルに付加することができます。
- セクションに実体を持たないセクションやオーバレイセクションなどの特殊な属性を付加して処理することができます。
- アブソリュートオブジェクトファイルを作成する通常のリンクの他に、リロケータブルオブジェクトファイルを生成するインクリメンタルリンクをサポートしています。
- リンケージエディタの入力ファイルおよび出力ファイルはIEEE695フォーマットに準拠しています。

リンク処理の概略　リンケージエディタは以下の順序でリンク処理を行います。

(1) アセンブラ(ASMxxシリーズ)により出力された複数個のモジュール中のセクションを、ユーザーが記述したコマンド言語による指示に従って編集結合します。
入力ファイル中のセクション(結合の対象となるセクション)を入力セクションと呼び、入力セクションを結合して作られた新しいセクションを出力セクションと呼びます。

(2) (1)で結合されたセクション(出力セクション)を、コマンド言語による指示どおりにメモリ上に配置します。

(3) 再配置可能な相対アドレスを持つコードやデータを絶対アドレスに変換し、アブソリュートオブジェクトファイルとして出力します。この際、ユーザーの指示により、C言語またはアセンブリ言語のソースレベルデバッグ用、アセンブリ言語のシンボリックデバッグ用のシンボル情報も出力します。
インクリメンタルリンクを行う場合は、相対アドレスの絶対アドレスへの変換は行われず、相対アドレスのままファイルに出力されます。インクリメンタルリンクにより作成されるファイルは、リロケータブルオブジェクトファイルです。

本書は「文法編」に当たるため、以下にコマンド言語の文法を中心に説明します。リンカの起動方法やオプションなどの説明は「操作編」を参照して下さい。

第2章 コマンド言語の概要

リンケージエディタには、リロケータブルオブジェクトをどのように結合･配置するかを指示するためのコマンド言語が用意されています。ユーザーは、このコマンド言語を用いることにより、リンケージエディタを細かく制御することができます。

2.1 コマンド言語の特長

リンケージエディタは、コマンド言語ファイル中に記述された、ターゲットマシンのメモリ構成、入力セクションの結合手順、結合したセクションをメモリに配置する位置などに従ってオブジェクトファイルのリンクを行います。
ユーザーは、コマンド言語ファイルを使用することにより、以下の機能を利用することができます。

- ターゲットマイクロプロセッサの使用可能メモリ領域の設定
- セクションを結合する順序の指定
- 異なる名前の同じ属性を持つセクションの結合
- セクションの任意の位置への配置
- オーバーレイセクションなどの特殊なセクション属性の指定
- 広域シンボルの定義、再定義

2.2 コマンド言語ファイルの構成

コマンド言語ファイルは、以下の 3 つの部分から構成されています。

- パラメータ指定部
  リンケージエディタでは固定オプションをコマンドファイルに記述できます。ここは、オプションを指定する部分です。
- メモリ定義部
  セクション命令部によって結合されたセクションをメモリ上にどう配置するかを指定する部分です。
- セクション命令部
  指定されたセクションをどう結合するかを指定する部分です。

必要のない部分は記述する必要はありません。このほか、コメントやシンボル値代入文を、コマンドファイルの任意の位置に記述することができます。
以下にコマンド言語ファイルの記述例を示します。

```
-la -o outfile.abs                    /* パラメータ指定部 */
objfile1.rel
objfile2.rel
table.rel

worksize = 0x0400;                    /* 代入文 */

MEMORY {                              /* メモリ定義部 */
   io(RW)   : org=0x0    len=0x100
   data.m   : org=0x100  len=0x380
   code     : org=0x8000 len=0x8000
}

SECTIONS {                            /* セクション命令部 */
   out_io   : {*(data_s_sec)} > io
   out_data : {objfile1.rel(data_sec1)
                     objfile2.rel(data_sec2)}
   out_code org=0x8200 : {table.rel(code_sec)}
}
```

パラメータ指定部　リンケージエディタの起動時のオプションパラメータとリンクするファイルのファイル名を記述します。パラメータ指定部では、オプションを使って出力ファイル名の指定やリンク情報の出力の指定をしたり、リンク処理の対象となるリロケータブルオブジェクトファイル名を指定します。オプションについての詳細は「C,C-Like,アセンブラ取扱説明書[操作編]」を参照して下さい。
パラメータとしては、"-V", "-$" と "-H" 以外のオプションと、ファイルリストが指定できます。また、パラメータは複数行に分けて記述することができます。
通常、パラメータ指定はコマンド言語ファイルの先頭に記述します。実際には、メモリ定義部、セクション命令部以外の任意の位置に、パラメータ指定を記述することができます。

メモリ定義部　メモリ定義部はMEMORY文で始まり、{　} で囲んで、メモリの定義を記述します。

```
MEMORY {
    <メモリ定義>        /* 一つのメモリ領域を定義 */
        :
    <メモリ定義>
}
```

メモリ定義部には、ターゲットマシン上で使用できるメモリ空間のサイズと属性の定義を記述します。このメモリ空間にセクションを配置し、絶対アドレスを決定することにより、アブソリュートオブジェクトファイルが作成されます。
メモリ定義の詳細については「第 3 章 メモリ定義部」で説明します。

セクション命令部　セクション命令部はSECTIONS文で始まり、複数のセクション命令の全体を{　} で囲みます。

```
SECTIONS {
    <セクション命令>    /* ｾｸｼｮﾝの結合を指定し､1つのｾｸｼｮﾝを作成 */
        :
    <セクション命令>
}
```

通常、セクションはコマンドラインで指定された順に結合処理され、絶対アドレスへの割り当てが行われますが、セクション命令を使用することにより、セクションの結合順序、配置位置などを自由に指定できます。
セクション命令の詳細については「第 4 章 セクション命令部」で説明します。

2.3 コマンド言語の基本要素

2.3.1 識別子

識別子は、英字、数字、特殊文字($　.　_)の任意の組み合わせで構成されます。識別子の最初の文字は英字または特殊文字でなければなりません。識別子は 32 文字目まで識別されます。
また、識別子は予約語、既定義シンボル、ユーザー定義シンボルに分類されます。

予約語　コマンド言語では、以下に示す単語が予約語として登録されています。ユーザーは、予約語をユーザー定義シンボルとして使用することはできません。ただし、オブジェクトファイル中に存在するセクション名の場合はエラーにならず、そのまま処理されます。予約語は大文字、小文字のいずれで記述しても予約語として認識されます。

ADDR	ALIGN	COPY	DSECT	LEN
LENGTH	MEMORY	NEXT	NOLOAD	ORG
ORIGIN	OVERLAY	SECTIONS	SIZEOF	

既定義シンボル　既定義シンボル(既定義メモリ名)には以下に示すものがあります。これらはメモリ名として登録されており、ユーザーはメモリ名としてのみ使用できます。既定義シンボルは大文字、小文字のいずれで記述されても既定義シンボルとして認識されます。以後、これらの既定義シンボルを既定義メモリ名と呼びます。

CODE	DATA	ROMDATA	
CODE.S	DATA.S	ROMDATA.S	small属性を持つ領域
CODE.M	DATA.M	ROMDATA.M	medium属性を持つ領域
CODE.L	DATA.L	ROMDATA.L	large属性を持つ領域

ユーザー定義シンボル　セクション名、ユーザー定義ラベルなど、アセンブラで記述したシンボルは全てユーザー定義シンボルです。この他に、コマンド言語中でシンボルを定義して値を設定することも可能です。
セクション名、ユーザー定義ラベルなどアセンブラで記述したシンボルは、原則としてそれらの処理系の制限を受けます。しかし、リンケージエディタはアセンブラに照らし合わせたシンボルの正当性チェックは行いません。すなわち、アセンブラで本来記述できないシンボルがコマンド言語中に含まれていても、リンケージエディタはそれについてのエラーを表示しません。ユーザー定義シンボルは、大文字と小文字が区別されます。

2.3.2 表現式

表現式は数値定数、シンボル、ロケーションカウンタリファレンスなどの項、およびこれらを演算子で結合したものです。

絶対式と相対式　表現式の評価結果は、"絶対" または "相対" の属性を持ちます。この属性は次の規則に従います。

- 絶対項のみからなる式の評価結果は "絶対" です。
- 1個の相対項と0個以上の絶対項から成る式の評価結果は、"相対" になります。
- 同じセクション内の相対項の減算結果は、"絶対" になります。
- 2個以上の相対項の評価は、上記演算を除いてできません。

数値定数

数値はそれ自身で固定の値を持ち、数字で始まる英数字列で表されます。全ての数値はリンカの内部では符号無しの倍長整数(32bits)として扱われ、それ以上の溢れは無視されます。また、実数は扱いません。

数値定数の表記法には 2 種類あります。一方は UNIX(C 言語)記法で、他方は IEEE 第 2 記法です。特に指定のない場合、数値は UNIX 記法と見なされます。IEEE 第 2 記法を使用する場合は、コマンド入力時にオプションスイッチ "-H" を指定してください。一回のリンク作業に使用するコマンドファイル中で、両記法の混在は許されません。

UNIX記法　2進数　：0y(ｾﾞﾛ､ﾜｲ)で始まる0または1の数字列で表します。
8進数　：0(ｾﾞﾛ)で始まる0～7の数字列で表します。
10進数　：0(ｾﾞﾛ)、0x(ｾﾞﾛ､ｴｯｸｽ)または0y(ｾﾞﾛ､ﾜｲ)以外で始まる0～9の数字列で表します。
16進数　：0x(ｾﾞﾛ､ｴｯｸｽ)で始まる0～9,A～F,a～fの英数字列で表します。

IEEE第2記法　2進数　：Bまたはbで終了する0または1の数字列で表します。
8進数　：O(ｵｰ),o,Qまたはqで終了する0～7の数字列で表します。
10進数　：Dまたはdで終了する0～9の数字列、または単に0～9の数字列で表します。
16進数　：0～9で始まりHまたはhで終了する0～9,A～F,a～fの英数字列で表します。

記法比較　両記法の比較を 表2.1 に示します。

表2.1 数値定数記法比較

	UNIX記法	IEEE第2記法
2進数	0y10010011　0y1111	10010011B　1111b
8進数	072541　0312　061134237　011	72541Q　312O　61134237o　11q
10進数	100　65535　12345	100D　65535d　012345
16進数	0x7FFF　0X154498　0xab5d	7FFFH　154498h　0ab5dh

ロケーションカウンタ

セクション命令部内に記述されるピリオド(.)は、リンケージエディタのロケーションカウンタの現在の値を示します。リンケージエディタがセクションを使用可能メモリ領域に配置するたびに、"." の値が更新されます。
ロケーションカウンタは、通常は、最後に配置したセクションのメモリ領域の次のアドレスを示しています。
後述の代入文を用いると、ロケーションカウンタの値を調節することができます。ただし、ロケーションカウンタの値を増やすことはできますが、減らすことはできません。

演算子

演算子は特殊文字により記述されます。演算子は C 言語の演算子と同じ意味を持ちます。使用可能な演算子とその優先順位を表 2.2 に示します。

表2.2 演算子の優先順位

優先順位	演算子
1	括弧((,))
2	単項演算子(!,",+,-)
3	2項乗余算(*,/,%)
4	2項加減算(+,-)
5	シフト(<<,>>)
6	比較(<,>,<=,>=)
7	比較(==,!=)
8	ビットAND(&)
9	ビットOR(|)
10	論理AND(&&)
11	論理OR(||)
12	代入演算子(=,+=)

特殊な演算子

表 2.2 に示した演算子以外に特殊な意味を持つ演算子があります。
これらはセクション命令の出力指定フィールドでのみ使用され、定義済みのセクションや数値定数に対する演算を行います。演算子は、

sizeof　addr　org　next　align

の5種類で演算子に続く ( ) 内に被演算項を伴って指定します。( ) の中の項には、演算時に値の確定しているものしか記述できません。

addr演算子

addr(<セクション名>)

- <セクション名>で指定した定義済みの出力セクションの開始アドレスが返されます。
- 指定されたセクションの開始アドレスが求められない場合は、エラーとなります。

```
SECTIONS  {
    DATASEC org=0x200                : {*(data_sec)}
    CODESEC org=addr(DATASEC)+0x200 : {*(code_sec)}
       /* 0x400番地にCODESECを配置 */
}
```

align演算子

align(<式>)

- 代入文中で指定した場合は、現在のロケーションカウンタを<式>の値でアライメントをとった値が返され、出力セクションのアライメント指定で指定した場合は、出力セクションのアライメント調整の指定となります。
- <式>の値は2のべき乗でなければなりません。
- <式>で指定した値の倍数番地でロケーションカウンタより大きくかつ最も近いアドレスを返します。
  例えば、ロケーションカウンタが0x127のとき、
  align(0x10)
  の値は0x130となります。

```
SECTIONS  {
    OUTSEC1 org=0 len=0x380  : {*(code_sec)}
    OUTSEC2 org=align(0x100) : {*(data_sec)}
       /* 0x400にOUTSEC2を配置 */
}
```

next演算子

next(<数値定数>)

- <数値定数>で指定した値の倍数のアドレスで、有効メモリ中のセクションが未配置のアドレスである値が返されます。
  例えば、メモリが0番地からあり、0x100から0x327のメモリにセクションが配置されている場合、
  next (0x10)
  next (0x100)
  の値はそれぞれ、0x10、0x400となります。

```
SECTIONS  {
    OUTSEC1 org=0 : {*(code_sec)}
    OUTSEC2 org=next(0x10) : {*(data_sec)}
    /* OUTSEC1のｾｸｼｮﾝのｻｲｽﾞが0x127の場合, */
    /* OUTSEC2は0x130に配置される */
}
```

org演算子

org(<セクション名>)

- <セクション名>で指定した定義済みの出力セクションの配置アドレスが返されます。
- 指定した定義済みセクションの配置アドレスが求められない場合は、エラーとなります。つまり、指定したセクションに対しては、配置アドレス指定（org=＜式＞）を使用して、予めアドレスを指定しておく必要があります。

```
SECTIONS  {
    CODESEC org=0x8000 : {*(code_sec)}
    DATASEC org=org(CODESEC)+0x200 : {*(data_sec)}
       /* 0x8200にDATASECを配置 */
}
```

sizeof演算子

```
sizeof( <セクション名> )
```

- <セクション名>で指定した定義済みの出力セクションのバイト単位の大きさが返されます。
- 指定した定義済みセクションのサイズが求められない場合は、エラーとなります。

```
SECTIONS  {
    OUTSEC1 org=0x1000 len=0x280 : {*(code_sec)}
    OUTSEC2 org=addr(OUTSEC1)+sizeof(OUTSEC1) :
                                  {*(data_sec)}
        /* 0x1280にOUTSEC2を配置 */
}
```

2.3.3 代入文

広域シンボルを定義したり、再定義する際に代入文を使用します。

【記述形式】

```
<ユーザー定義シンボル>  =   <表現式>;
<ユーザー定義シンボル>  +=  <表現式>;
```

- 代入文の最後にはセミコロン(;)が必要です。
- 代入文は、セクション命令部内でも使用できます。リンケージエディタは、セクションの割り当てを行っていく際に、記述された順序で表現式を評価し、その値をシンボルに代入します。
- セクション命令部で代入文を使用することにより、ロケーションカウンタの値を調節することができます。ただし、ロケーションカウンタの値を減らすことはできません。

【例】

```
/* 代入文によるロケーションカウンタの調節の例 */
OUTSEC:  {
    . += 0x10;          /* 0x10バイトの空き領域ができる */
    file1.rel(SECTION1)
    . = align(4);       /* 4の倍数のアライメントをとる */
    file2.rel(SECTION1)
}
```

2.3.4 コメント

コマンド言語ではコメントが記述できます。C 言語と同様に、/*　*/ で囲まれた部分がコメントとなります。コメントのネストはできません。なお、コメントには日本語が記述できます。また、コメントは、複数行にまたがって記述できます。

第3章 メモリ定義部

メモリ定義部は、ターゲットマシン上で使用するメモリ空間のサイズと属性を定義します。リンケージエディタは、このメモリ空間にセクションを配置し、アブソリュートモジュールを作成します。
メモリ定義部は MEMORY 文に続いて { }で囲んで複数のメモリ定義を記述します。

3.1 メモリ定義の記述形式

メモリ定義の記述形式は、以下の通りです。

【記述形式】

```
<メモリ名> [<属性>] : org=<アドレス> [,] len=<サイズ>
```

<メモリ名> 使用可能メモリ領域を定義し、その領域に名前をつけます。メモリ名には、既定義メモリ名またはユーザーが定義したメモリ名を記述します。ユーザー定義のメモリ名の場合は、名前は32文字以内で記述します。

メモリ名は、リンケージエディタの内部だけで使用されますので、出力ファイルには渡されません。

<属性> 使用可能メモリ領域の属性を ( ) で囲んで指定します。

R 読み出し可能

W 書き込み可能

X 実行可能

I 初期化可能

これらは、小文字で指定しても構いません。

指定を省略すると、R、W、X、Iの全ての属性を持っているものとみなされます。

リンケージエディタは、Iの属性を指定したメモリ領域に配置されたセクション内に空き領域ができると、その空き領域を初期化します。初期化に関しては、「4.6 パディングフィールド」で説明します。

<アドレス> 使用可能メモリ領域の先頭アドレスを指定します。32ビットの符号無し整数で指定します。

orgの代わりにoriginを使って、origin=<アドレス>という指定も可能です。

<サイズ> 使用可能メモリ領域のサイズをバイト単位で指定します。32ビットの 符号無し整数で指定します。

lenの代わりにlengthを使って、length=<サイズ>という指定も可能です。

アドレスとサイズの指定の間は、コンマまたは空白で区切ります。アドレスとサイズの指定の順番を入れ替えることはできません。

【例】

```
MEMORY {
    io(RW)   : org=0x0       len=0x100  /*ユーザー定義メモリ名*/
    data.m   : org=0x100     len=0x380
    code     : origin=0x8000 length=0x8000
}
```

3.2 メモリ定義の意味

リンケージエディタは、コマンド言語ファイル内にメモリ定義が存在すると、そこで定義されていないメモリ空間は存在しないものとみなし、セクションに絶対アドレスを割り当てる際にエラーチェックを行います。
そのため、実アドレスが割り当てられる出力セクション(DSECT、COPY 以外の属性を持つセクション)は存在しないメモリ空間に割り当てることはできません。
出力セクションおよび DSECT、COPY 属性については、「4.4 属性フィールド」を参照してください。

メモリ定義がない場合　メモリ定義がコマンド言語ファイル内に存在しない場合、リンケージエディタはターゲットマシンの全メモリ空間が有効で、メモリ属性は全て有効であるとみなします。従って、リンケージエディタはセクションの絶対アドレスへの割り当てを0番地からセクション命令で指定された順に行います。

属性の対応　アセンブリ言語のsection命令で指定したセクションの種類とメモリ定義でのメモリ属性との関係は表3.1のようになります。

表 3.1 セクションの種類と必要メモリ

セクションの種類	必要メモリ属性
CODE	RX
DATA	RW
ROMDATA	R

リンケージエディタは、セクションをメモリ空間に配置する際に、表 3.1 の対応のチェックを行い、対応するメモリ属性が指定されてない場合はエラーとなります。例えば、CODE セクションを配置するメモリには、R(読出し可能)と X(実行可能)の属性が必要です。
セクションを配置するメモリには必ず R 属性が必要です。セクションを配置するメモリに R 以外の属性を指定する場合は、RW、RX のように、必ず R も合わせて指定してください。メモリ属性の指定を省略した場合は、RWXI が指定されたものとみなされるのでエラーは生じません。

3.3 既定義メモリ名

既定義メモリ名とその意味は、次の通りです。

code[.サイズ] 機械命令を配置する領域です。code領域はRX属性を持ちます。アセンブラソースファイルのsection命令の種類でcodeが指定されたセクションがこのメモリ領域に割り当てられます。

data[.サイズ] プログラム実行時に値を変更するデータを配置する領域です。data領域はRW属性を持ちます。アセンブラソースファイルのsection命令の種類でdataが指定されたセクションがこのメモリ領域に割り当てられます。

romdata[.サイズ] プログラム実行時に値を変更しない定数を配置する領域です。romdata領域はR属性を持ちます。アセンブラソースファイルのsection命令の種類でromdataが指定されたセクションがこのメモリ領域に割り当てられます。

サイズはピリオド(.)に続けて"S", "M", "L"のいずれかを記述します。
通常それぞれの既定義メモリ名に ".サイズ" をつけて、アセンブラソースファイルの section 命令のサイズに対応させます。このとき、ソースプログラムで指定されたセクションの種類と、割り当てられた既定義メモリの属性が合っていなければなりません。相対番地セクションではセクションのサイズ、種類が一致しなくてはならず、絶対番地セクションではセクションの種類が一致しなければなりません。

既定義メモリ名の効果

メモリ名に既定義メモリ名を用いると、後述するセクション命令部で特に指定しない限り、アセンブリ言語ファイルの section 命令で指定された種類と配置に従い、対応するメモリ名の領域に自動的に配置されます。すなわち、アセンブリ言語ファイルの medium 属性を持つ code セクションは、メモリ定義部で定義した code.m 領域に配置されます。
アセンブラの配置属性 small, medium, large は各 CPU によりサイズが異なりますので、既定義メモリのアドレス指定時は注意が必要です。必ず矛盾の無いように設定して下さい。例えば、TLCS-900 の medium は 16 ビットですので、リンケージエディタのメモリ定義では、code.m, data.m, romdata.m は 0 から 0xffff の間に無ければなりません。各 CPU と配置属性の関係は「第 1 部 アセンブリ言語」の section 擬似命令の説明を参照して下さい。

既定義メモリ名のサイズ

ユーザー定義のメモリではアドレスを重ねて定義することは出来ませんが、既定義メモリでは同じ種類の異なるサイズのメモリに限り、アドレスを重ねて定義することが出来ます。例えば、メモリ定義部で定義した code.s 領域に「code small」を配置し、code.s 領域が余っていたら、small のすぐ後ろから、「code medium」を配置します。

第4章 セクション命令部

セクション命令部では、どの入力セクションどうしを、どのような順番で結合するか、結合した結果のセクションをどのメモリ領域に配置するかを指定します。
コマンド言語ファイルにセクション命令部を記述しない場合は、セクションは入力の指定順に結合され配置されます。
セクション命令部は SECTIONS 文で始まり、{ } で囲んでセクション命令を複数記述します。セクション命令部にはこの他に、代入文(広域シンボルの定義や再定義)が記述できます。

4.1 セクション命令の記述形式

個々のセクション命令の記述形式は、以下のようになります。

【記述形式】

```
<出力セクション名> [<出力指定>] [<属性>] :
        {<入力セクション指定>} [ = <パディング>] [ > <出力メモリ>]
```

<出力セクション名> 結合後のセクション名を指定します。入力セクションと同じ名前も指定できます。

<出力指定> 出力セクションの配置アドレス、アライメント、サイズなどを指定します。 この指定は省略可能です。

<属性> 出力セクションに特殊な属性を付加します。この指定は省略可能です。

<入力セクション指定> 結合するセクション名を指定します。

<パディング> 入力セクション間の空き領域を初期化する数値を指定します。この指定は省略可能です。

<出力メモリ> 出力セクションをユーザー定義メモリ領域に配置することを指示し、メモリ定義で指定したユーザー定義メモリ名を指定します。この指定は省略可能です。

【例】

```
SECTIONS  {
   OUTSEC1                              /* 出力セクション名 */
      org=0 len=0x1000 addr=0x8000 /* 出力指定 */
      (NOLOAD) :                        /* 属性 */
      { file1.rel(code_sec) file2.rel(code_sec) }
                                        /* 入力セクション指定 */
      = 0xffff                          /* パディング */
      > MEM1                            /* 出力メモリ */

   OUTSEC2 : {*(code_sec)}
}
```

"OUTSEC1" から "> MEM1" までが一つのセクション命令です。これは、全てのフィールドが記述されている例です。

OUTSEC2で始まる行も一つのセクション命令で、この命令には、出力セクション名、入力セクション指定の最低限の指定しかありません。

以下ではセクション命令の各フィールドについて説明します。

4.2 出力セクション名フィールド

出力セクション名フィールドには、出力ファイルに出力するセクション名を指定します。入力セクション名と同じ名前も指定できます。
32 文字までのシンボルが指定できます。

4.3 出力指定フィールド

出力指定フィールドには、出力セクションの配置アドレス、アライメント、サイズ、開始アドレスを指定します。ただし、配置アドレスとアライメントを同時に指定することはできません。

【記述形式】

```
[<配置アドレス>][<アライメント>][<サイズ>][<開始アドレス>]
```

<配置アドレス>　出力セクションを配置するメモリアドレスを指定します。
<アライメント>　出力セクションの配置を調整するアライメント境界を指定します。
<サイズ>　出力セクションのサイズの最小値を指定します。
<開始アドレス>　開始アドレスとは、シンボルに絶対アドレスを割り付ける際の基準となるアドレスです。実行中に他の領域へ転送して実行するようなコード領域やデータ領域を作成するときなど、オブジェクトを格納しておく領域と実行時の領域が異なる場合に、実行時のアドレスを開始アドレスとして指定します。

それぞれの指定を省略することも可能です。配置アドレスを指定せずに、割り当てるメモリ名を指定することも可能です。出力メモリの指定に関しては
「4.7 出力メモリフィールド」を参照してください。

4.3.1 配置アドレス指定

【記述形式】

```
org = <式>
```

【説明】

- <式>の値には出力セクションのメモリ配置アドレスを指定します。配置アドレスを指定すると、出力セクションは固定アドレスを持ちます。
- 本指定を省略した場合、配置アドレスと開始アドレスが同じになります。
- 入力セクションのサイズ属性(small,medium,large)と、配置アドレスが該当するメモリ定義のメモリのサイズ属性(.s,.m,.l)が矛盾する場合は、ワーニングとなります。しかし、セクションの種類(code,data,romdata)は一致しなくてもかまいません。
- 配置アドレスとアライメントを同時に指定することはできません。

【例】

```
SECTIONS  {
   OUTSEC1 org = 0x8000 : {objfile1(code_sec)}
                /* 0x8000 番地から配置 */

   OUTSEC2 org = addr(OUTSEC1)+0x200 : {objfile2(code_sec)}
                /* 0x8200 番地から配置 */
}
```

4.3.2 アライメント指定

【記述形式】

```
align = <式>
```

【説明】

- <式>で指定した値の倍数番地が出力セクションの配置アドレスとなります。
- <式>の値は2のべき乗でなければなりません。
- 開始アドレス指定と同時に指定しない場合は、出力セクションの配置アドレスと開始アドレスは同じになります。
- セクションの属性とアライメントされた結果の配置アドレスがメモリ定義の指定に矛盾する場合は、エラーとなります。
- 配置アドレスとアライメントを同時に指定することはできません。

【例】

```
MEMORY  {
    MEM1 : org=0x180 , len=0x480
}
SECTIONS  {
    OUTSEC align=0x100 : {objfile(code_sec)} > MEM1
        /* 0x100でアライメントをとって0x200番地に配置 */
}
```

4.3.3 サイズ指定

【記述形式】

```
len = <式>
```

【説明】

- <式>の値を出力セクションのサイズの最小値に指定します。
- <式>の値が入力セクションの合計サイズ以上の場合は、出力セクションのサイズは<式>で指定した値になります。
- <式>の値が入力セクションの合計サイズより小さい場合は、ワーニングとなり、出力セクションのサイズは入力セクションの合計サイズとなります。
- 配置アドレスの指定と同時に指定できます。この場合は配置アドレス指定より後に指定してください。

【例】

```
SECTIONS  {
    OUTSEC1 org=0x100 len=0x100 : {objfile(data_sec)}
        /* OUTSEC1のセクションサイズが0x100未満の場合は、*/
        /* サイズは0x100になる。                          */
    OUTSEC2 len=0x300 : {objfile(code_sec)} > MEM
        /* OUTSEC2のセクションサイズが0x300未満の場合は、*/
        /* サイズは0x300になる。                          */
}
```

4.3.4 開始アドレス指定

【記述形式】

```
addr =<式>
```

【説明】

- <式>の値を出力セクションの開始アドレス(実行アドレス)に指定します。
- 出力セクションの配置アドレスと開始アドレスを別にしたい場合に指定します。配置アドレスと開始アドレスの両方を指定した場合、開始アドレスから始まる領域にシンボル情報、配置アドレスから始まる領域にオブジェクトデータが出力されます。これは、実行中に他の領域に転送して使用するようなコードやデータがある場合に使用します。そのとき、配置アドレスには転送もとのアドレスを、開始アドレスには転送先のアドレスを指定します。
- 本指定を省略した場合、配置アドレスと開始アドレスは一致します。
- 出力セクションの種類(code,data,romdata),サイズ属性(small,medium, large)と開始アドレスが該当するメモリ定義のメモリの種類,サイズ属性(.s,.m,.l)が矛盾する場合は、ワーニングとなります。
- 開始アドレスもメモリ定義で定義されている領域内になければなりません。
- 開始アドレスは配置アドレスより後に指定します。

【例】

```
SECTIONS {
    OUTSEC org=0x10000 addr=0x4000 : {objfie(data_table)}
}
```

OUTSEC セクションは、出力オブジェクトとしては 0x10000 番地に配置されています。しかし、シンボル情報は、0x4000 番地に配置されます。例えば OUTSEC セクションの先頭が

```
TABLE    db   0x10
```

という命令であった場合、オブジェクトファイル中ではデータ(10)は 0x10000 番地に配置されていますが、変数名 TABLE の値(アドレス)は 0x4000 番地になります。

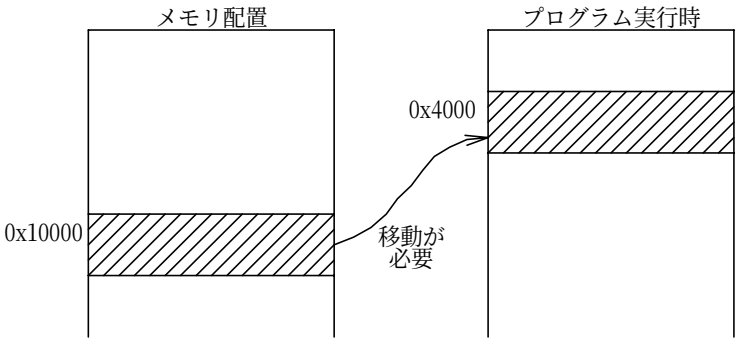


図 4.1: 開始アドレス指定

4.4 属性フィールド

出力セクションに特殊な属性を付ける場合に指定します。指定できる属性は、DSECT、COPY、NOLOAD、OVERLAYの4種類です。これらの属性を持つセクションと通常のセクションの違いを表4.1に示します。

通常の使い方をするときは、出力セクションに属性を付ける必要はありません。

表 4.1: 特殊な属性を持つセクションの特徴

属性	オブジェクトデータ	オーバーラップ	アドレス
なし(通常)	出力する	できない	実アドレス
DSECT	出力しない	できる	仮想アドレス
COPY	出力する	できる	仮想アドレス
NOLOAD	出力しない	できない	実アドレス
OVERLAY	出力する	できる	実アドレス

4.4.1 DSECT属性

【記述形式】

(DSECT)

【説明】

- 出力セクションに実体を持たないダミーセクション属性を指定します。DSECT属性は実際にメモリが割り当てられないため、他のセクションとオーバーラップできます。また、メモリ定義で宣言されていないメモリ領域(仮想アドレス)に配置することができます。
- ダミーセクションに対して配置アドレスを指定しない場合、仮想的に0番地に配置されます。
- 出力セクションが持つもともとのセクション属性(CODE、DATAなどの種類)は、DSECT属性が指定された時点で意味を持たなくなります。
- ダミーセクションのオブジェクトデータは出力されませんが、シンボルなどのデバッグ情報はオブジェクトファイルに出力されます。
- DSECTセクション内で定義されている広域シンボルは再配置され、出力ファイルのシンボルテーブルに登録されます。
- DSECTセクション内の未定義の外部シンボルに対して、ライブラリファイルが検索され、通常のセクションと同様にリンクされます。
- ダミーセクションは実体がなく、メモリに割り当てられないため、<出力メモリ>は指定できません。

【例】

```
SECTIONS  {
    OUTSEC1 (DSECT) : {objfile(INSEC1)}
        /* 実体なし。仮想的に0番地から配置 */
    OUTSEC2 org=0x600 (DSECT) : {objfile(INSEC2)}
       /* 実体なし。仮想的に0x600番地から配置   */
       /* 0x600がメモリ定義されていなくてもよい */
}
```

4.4.2 COPY属性

【記述形式】

(COPY)

【説明】

- COPY属性は、オブジェクトデータを出力する点を除いてDSECT属性と同じです。COPY属性を指定すると、オブジェクトデータは出力されますが、実際にはメモリに割り当てられず、もともとのセクション属性は無視されます。また、配置アドレスを指定しなければ、仮想アドレスは0番地となります。

【例】

```
SECTIONS  {
    OUTSEC1 (COPY) : {objfile(INSEC1)}
        /* 実体あり。仮想的に0番地から配置 */
    OUTSEC2 org=0x600 (COPY) : {objfile(INSEC2)}
        /* 実体あり。仮想的に0x600番地から配置   */
        /* 0x600がメモリ定義されていなくてもよい */
}
```

4.4.3 NOLOAD属性

【記述形式】

(NOLOAD)

【説明】

- オブジェクトデータが出力されない点を除いて、通常のセクションと同じです。NOLOAD属性を指定すると、オブジェクトデータは出力されませんが、メモリ割り当ては実アドレス空間に割り当てられます。メモリマップにも出現します。

【例】

```
MEMORY  {
    MEM : org=0x1000 len=0x1000
}
SECTIONS  {
    OUTSEC1 (NOLOAD)  : {objfile(INSEC1)} > MEM
    OUTSEC2           : {objfile(INSEC2)} > MEM
}
```

OUTSEC1 は通常のセクションと同様に 0x1000 から配置されますが、実際のオブジェクトデータは出力されません。OUTSEC2 は通常のセクションです。OUTSEC1 のサイズが 0x0800 であれば OUTSEC2 は 0x1800 から配置されます。OUTSEC1 のオブジェクトデータは出力されませんので、0x1000 から 0x1800 までの領域は空領域となります。

4.4.4 OVERLAY属性

【記述形式】

```
(OVERLAY)
```

【説明】

- OVERLAY属性を持ったセクションは、通常のセクションと同様に配置され、オブジェクトデータが出力されます。通常のセクションと異なる点は、他のセクションとオーバーラップ可能な点です。通常のセクションは、既に他のセクションが配置されている領域に配置することはできませんが、OVERLAYセクションは、既に他のセクションが配置されている領域に配置することが可能です。
- OVERLAYセクションには必ず配置アドレスを指定しなければなりません。

【例】

```
SECTIONS  {
   OUTMAIN org=0x9000             : {objfile(code1)}
   OUTSEC1 org=0x9000 (OVERLAY)   : {ovlfile1(code2)}
   OUTSEC2 org=0x9000 (OVERLAY)   : {ovlfile2(code3)}
   OUTSEC3 org=0x9000             : {objfile(code4)}
      /* error, ignored */
}
```

OUTSEC1 と OUTSEC2 は OVERLAY 属性を持つセクションなので、既に OUTMAIN セクションが配置されている 0x9000 からのメモリ上に配置することができます。

OUTSEC3 は通常のセクションなので、0x9000 からのメモリに配置しようとすると、既に OUTMAIN セクションが配置されているためエラーとなります。

4.5 入力セクション

入力セクションフィールドでは、リンケージエディタが結合する複数ファイル内のセクションの集合を指定します。「{　}」で囲んで、複数のファイル名とそのファイルに含まれるセクションを指定します。

入力セクション指定

```
{ <ファイル名>( <セクション名>... )
    :
  <ファイル名>( <セクション名>... ) }
```

【説明】

- ❑ <ファイル名>で結合するセクションを含むリロケータブルオブジェクトファイルを指定します。
- ❑ コマンドラインやコマンド言語ファイルのパラメータ指定部で指定していない ファイル名を直接<ファイル名>に記述した場合、このファイルも入力ファイルの対象となります。
- ❑ <ファイル名>の代わりにアスタリスク「*」を指定することができます。この場合、コマンド言語ファイルのパラメータ指定部やコマンドラインで指定した、全てのリロケータブルオブジェクトファイルとライブラリファイルが指定されたものとみなされます。
- ❑ <セクション名>で結合する1つまたは複数のセクション名を指定します。すると、入力セクション列が1つの出力セクションにまとめられます。このとき、入力セクションのセクション属性(CODE、DATAなどの種類属性)は、同じでなければなりません。出力セクションは、DSECTなどの特殊な属性を指定しない限り、入力セクションと同じ属性を持ちます。
- ❑ <セクション名>では、同一ファイル内にある複数のセクションを指定することができます。複数のセクションを指定する場合は、<セクション名>をスペースで区切って指定します。
- ❑ <セクション名>を省略した場合、<ファイル名>で指定したファイル中の全てのセクションが結合の対象となります。

【例】

```
SECTIONS  {
    OUTSEC1 : {
        file1.rel(data_sec)
        file3.rel(data_sec)
            /* 指定したファイルのdata_secセクションを結合 */
    }
    OUTSEC2 : { file2.rel }
            /* file2.relの全セクションを結合 */
    OUTSEC3 : { file3.rel(vector const) }
            /* file3.relのvectorとconstセクションを結合 */
    OUTSEC4 : { *(data_sec) }
            /* 未結合の全てのdata_secセクションを結合 */
}
```

4.6 パディングフィールド

リンケージエディタは、入力セクションを結合する際に生じる入力セクション間の空白領域を初期化します。これをパディングといいます。この場合、初期化する値を指定することができます。

パディング指定

```
=<数値>
```

【説明】

- <数値>で入力セクションを結合後の境界領域を初期化する値を指定します。
- <数値>には2バイトの数値を指定します。
- パディングの開始アドレスが奇数、偶数に関わらず、パディングの開始アドレスから上位バイト、下位バイトの順に初期化されます。すなわち、パディング領域が奇数バイトの場合、最後の1バイトは上位バイトで初期化されます。
- パディングの指定を省略した場合、リンケージエディタの起動時オプションの "-F" で指定したパディング値となります。また、"-F" オプションも省略した場合は、0でパディングが行われます。
- パディングを行うのはI属性をもつメモリ領域のみで、I属性を持たないメモリ領域に対して指定した場合、エラーとなり、パディングの指定は無視されます。

【例】

```
SECTIONS  {
    OUTSEC org = 0x1000 :
      file1.rel(sub1)
      . += 0x100;   /* この0x100バイトの領域がパディングされる */
      file1.rel(sub2)
    } = 0xffff      /* 0xffffはパディング値 */
}
```

4 行目の ". += 0x100;" は代入文で、ファイル file1.rel の sub1 セクションを配置した後にロケーションカウンタを 0x100 だけ進めます。

例えば sub1 セクションのサイズが 0x800 のとき、OUTSEC セクションの配置アドレスは 0x1000 番地ですから、配置後のロケーションカウンタの値は 0x1800 となり、sub2 セクションは 0x1900 から配置されます。

ここで 0x1800 から 0x18ff までの領域はセクション間の空き領域ですので、この領域が 0xffff で埋められます。

4.7 出力メモリフィールド

出力メモリフィールドは、メモリ定義部で定義したユーザー定義メモリ領域に出力セクションを割り当てるように指示します。この割り当ての際にメモリ領域とセクションの RWXI 属性の照合が行われ、矛盾が生じた場合はエラーとなります。

出力メモリ指定

```
>><メモリ名>
```

【説明】

- <メモリ名>で指定したユーザー定義メモリ領域に出力セクションが配置されます。
- <メモリ名>はメモリ定義指定部で定義したユーザー定義メモリ空間の名前です。
- セクション命令部の指定とメモリ定義部の指定に矛盾が生じた場合は、エラーとなります。
- 出力メモリ指定で既定義メモリ名を指定してもエラーにはなりません。ただし、既定義メモリへの配置はセクションの種類によって自動的に行われるので、出力メモリ指定は特に意味を持ちません。

【例】

```
MEMORY  {
    MEM1 : org=0x0000 len=0x1000
    MEM2 : org=0xf000 len=0x1000
}
SECTIONS  {
    RAMSEC : {file1(data)} > MEM1    /* RAMSECをMEM1に配置 */
    ROMSEC : {file1(code)
       file2(code)} > MEM2 /* ROMSECをMEM2に配置 */
}
```

4.8 セクション命令部の省略

セクション命令部を省略した場合、リンケージエディタはコマンドラインやコマンド言語ファイルのパラメータ指定部で指定された順にセクションのリンク作業を行います。このとき、同じ名前を持つセクションは連続してリンクされます。ただし、同じ名前のセクションはすべての入力ファイルを通して同じ属性でなければなりません。

第5章 リンク処理

5.1 リンク処理の手順

リンケージエディタは以下の手順でリンク処理を行い、アブソリュートオブジェクトモジュールを作成します。

(1) メモリ状態の認識
メモリ定義部の記述から、物理的なメモリ状態を認識します。

(2) 出力セクションの作成
セクション命令部の記述に従って、入力セクションを結合し、出力セクションを作成します。
通常は複数の出力セクションが作成されます。

(3) 出力セクションのメモリへの配置
全ての出力セクションを、メモリ定義部とセクション命令部の記述に従ってメモリ上に配置します。

(4) 相対アドレスの絶対アドレスへの変換
メモリ領域に配置されて決定したアドレスに基づいて、オブジェクト中の相対項を絶対アドレスに再配置します。

以下で「(2)出力セクションの作成」、「(3)メモリへの配置」の処理の詳細を示します。

5.2 出力セクションの作成

出力セクションの作成は以下の手順で行われます。

(1) リンケージエディタは、入力セクションのうち、絶対番地セクションはそのまま出力セクションとし、相対番地セクションはコマンド言語ファイルの内容に従って結合処理を行い、出力セクションを作成します。

(2) 入力ファイルとして指定されたリロケータブルオブジェクトファイル中の全ての入力セクションに対して出力セクションを作成した後、外部参照シンボルの解決(未定義のシンボルの定義を探す)処理を行います。

(3) 外部参照シンボルの解決処理の結果未定義のシンボルが残った場合、入力ファイルとして指定されているライブラリファイルを検索し、未定義のシンボルが定義されているモジュールを探して、そのモジュール内の全てのセクションを新たな入力セクションとして(1)の処理を再開します。

(4) 未定義のシンボルがなくなるか、未定義シンボルに対応するシンボルの定義がライブラリファイル中に存在しなくなった時点で出力セクションの作成処理を終了します。

5.2.1 絶対番地セクションの扱い

絶対番地セクションは、すでに絶対アドレスが割り当てられているため、他のどのセクションとも結合されません。リンケージエディタは、それぞれの絶対番地セクションを単独のセクションとして扱い(セクション命令部の出力指定フィールドで配置アドレスを指定したセクションと同じ扱い)、入力セクションと同じセクション名、同じ属性(セクションの種類)、同じ配置アドレスの出力セクションを作成します。このとき、配置先のメモリ領域と絶対番地セクションで種類やRWXI 属性が異なる場合は、エラーメッセージが出力されますが、処理は続行されます。

セクション命令部で、入力セクションとして絶対番地セクションが指定されている場合は、エラーを出力し、指定を無視します。

入力セクション指定フィールド中に同名の絶対番地セクションと相対番地セクションが現れた場合は、エラーとなり、先に現れた(すでに出力セクションが作成された)セクションを有効とし、後に指定したセクションをリンク対象からはずします。リンク処理は続行します。

5.2.2 相対番地セクションの結合

結合するセクション

リンケージエディタは、以下の原則に従って相対番地セクションの結合を行います。

- リンケージエディタは入力セクションのうち相対番地セクションを、コマンド言語ファイルのセクション命令部に記述された順序で結合し、指定された名前の出力セクションを作成します。
- 同一名の入力セクションであっても、セクション命令部で異なる出力セクションに結合するよう指定されている場合は、セクション命令部の指定に従って別々の出力セクションに結合します。
- 出力セクションが指定されていない入力セクション(コマンドラインやパラメータ指定部でファイル名だけを指定した場合など)がある場合は、入力セクションと同じ名前の出力セクションを作成します。ただし、すでに同名の出力セクションが作成されている場合は、そのセクションに続けて結合します。
- セクション命令部が記述されていない場合は、指定された入力ファイルの順に、入力セクション名と同名の出力セクションを作成します。この結果、同名の入力セクションは連続して結合されます。

以上の結果、入力ファイル中のセクションは必ず出力セクションに出力されます。このとき、出力セクションの属性(種類、境界調整値)は、入力セクションと同じ属性になります。

ただし、同名のセクションであっても、結合するセクションの種類(CODE、DATA、ROMDATA)と境界調整値(アセンブラの section 命令の align で指定された値)が一致していないと結合できません。リンケージエディタエディタはエラーを出力し、後から指定されたセクションをリンクの対象からはずします。

セクションの結合処理　リンケージエディタは、入力セクションを連結結合して出力セクションを作成します。

- 入力セクションを連続したアドレスに、順番に結合します。配置の順番は、セクション命令部の入力セクション指定で指定された順序です。ただし、入力セクション指定で入力ファイル名が「*」で指定されている場合は、コマンドラインまたはコマンド言語ファイルのパラメータ指定部で指定された順序となります。
- 結合するセクション間の境界は、入力セクションの境界調整値に従って調節します。
- 相対セクションでは、セクション内のシンボルは、セクションの先頭からの相対アドレスが割り当てられています。リンケージエディタは、それぞれのシンボルの相対アドレスを、そのシンボルがあるセクションの先頭アドレスに加えることにより、結合後の相対アドレスを決定し、セクションをメモリ領域に配置するときに絶対アドレスを決定します。
- 出力セクションのサイズは結合前の各セクションのサイズと境界調整値で調整された境界のサイズの合計になります。

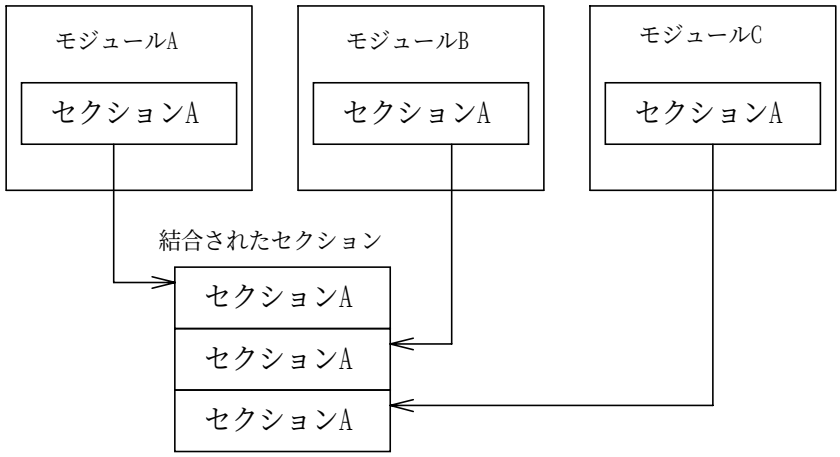


図5.1セクションの結合

5.2.3 シンボルの解決とライブラリファイルの結合

シンボルの解決　リンケージエディタの入力オブジェクトモジュール内でextern命令で宣言された外部参照シンボルは、モジュール内にその実体が存在せず、アセンブルの段階で値の決定が保留になったシンボルです。リンケージエディタは、この外部参照シンボルと同じ名前の外部定義シンボルを検索します。
対応する外部定義シンボルが複数発見された場合は、エラーとなります。

ライブラリの結合　全ての出力セクションに対してシンボルの解決処理を行なった結果、未解決の外部参照シンボルが残った場合、リンケージエディタはライブラリファイルから、そのシンボルを外部定義している(public命令で宣言している)モジュールを探し出します。
全ての未解決シンボルに対してライブラリファイルの検索を行い、未解決シンボルを定義している全てのモジュール中のセクションを新たな入力セクションとして結合し、出力セクションの作成処理を行います。
上記の過程を繰り返して、未解決シンボルが無くなった時点で出力セクションの作成処理は終了します。
ライブラリファイル中に未解決シンボルを定義しているモジュールが見つからなかった場合や、入力ファイルにライブラリファイルがない場合などシンボルが未解決のまま残ると、エラーメッセージを出力し、出力セクションの作成処理を終了します。

5.3 出力セクションのメモリ配置

メモリ配置は、メモリ定義部とセクション命令部の指定に従って行われます。メモリ定義部が省略されると、出力セクションを、セクション命令部の指定順に 0 番地から配置します。

メモリ配置の順序

セクションのメモリへの割り当ては以下の順序で行われます。

(1) アドレスが指定されている出力セクションの配置

- アドレスが指定されている出力セクションとは、絶対番地セクション(入力セクションがそのまま出力セクションとなります)と、配置アドレスまたは開始アドレスが指定されている出力セクションです。
- アドレスが指定されている出力セクションの配置アドレスに従ってセクションをメモリ領域に配置します。
- セクションの種類によって必要なメモリ属性が配置先のメモリにない場合、例えばセクションの種類がcodeであるセクションを配置するメモリにX属性がない場合、リンケージエディタはエラーを出力し、セクションを異なる属性のメモリに配置したまま処理を続行します。
- 絶対番地セクションの配置先が既定義メモリ領域の場合は、アセンブラソースファイルのセクションの種類(code,data,romdata)と既定義メモリの種類が同じであれば、既定義メモリのサイズに関わらず配置します。
- 配置アドレスが指定されたセクションの配置先が既定義メモリ領域の場合は、アセンブラソースファイルのセクションのサイズ(small,medium, large)と既定義メモリのサイズ(.s,.m,.l)が同じであれば、既定義メモリの種類に関わらず配置します。
- 開始アドレスが指定されたセクションの配置先が既定義メモリ領域の場合は、アセンブラソースファイルのセクションの種類とサイズが既定義メモリの種類とサイズと同じでなければなりません。
- 配置先がユーザー定義メモリ領域の場合は、セクションの種類によって必要なRWXI属性が配置先のメモリ領域にない場合、例えばセクションの種類がcodeであるセクションを配置するメモリにX属性がない場合、リンケージエディタはエラーを出力し、セクションを異なる属性のメモリに配置したまま処理を続行します。

(2) 出力メモリが指定されている出力セクションの配置

- セクション命令部で、配置アドレス(org)が指定されずに、<出力メモリ>が指定されている出力セクションを、メモリ領域中のセクションが配置されていない領域に配置します。
- 配置先が既定義メモリ領域の場合は、アセンブラソースファイルのセクションの種類,サイズと既定義メモリ名が一致していないとエラーになります。例えば、code mediumセクションを"data.m"という既定義メモリに配置した場合、またはcode mediumセクションを"code.l"という既定義メモリに配置した場合エラーになります。
- 指定のメモリ領域に相対番地セクションを配置できなかった時(セクションの種類によって必要なRWXI属性が配置先のメモリにない場合、セクションを配置しようとしたメモリが既に使用されていたり、定義外または使用不可能だった場合など)は、リンケージエディタはエラーを出力し、そのセクションは配置せず、次の(3)の処理で配置します。
- 指定されたメモリ領域にセクションを配置する余裕がない場合もリンケージエディタはエラーを出力し、そのセクションは配置せず、次の(3)の処理で配置します。

(3) 残りのセクションの配置

- 既定義メモリが定義されている場合は、各セクションを対応するメモリ領域に配置します。
- 上記の処理で配置されなかったセクションは、RWXI属性を満たすメモリ領域をメモリ定義部で定義した領域の先頭から検索し、配置します。
- RWXI属性を満たすメモリがない場合はエラーを出力し、未定義のメモリ領域の先頭から空き領域を検索し、強制的に配置します。

メモリ配置時のエラー　メモリ配置時には、以下のような場合にエラーが発生します。

- メモリ定義部でメモリ指定していない領域にセクションを配置しようとした。
- セクションの種類,サイズと異なる既定義メモリ領域にセクションを配置しようとした。
- アドレスが指定されている出力セクションがメモリ指定のない部分へ配置された。
- セクションが使用可能メモリ領域を越えて配置された。
- オーバーレイ属性を持たないセクションが同じアドレスに重複して割り当てられた。

5.4 インクリメンタルリンク

5.4.1 インクリメンタルリンク

リンケージエディタの起動時に "-r" オプションを指定した場合、リンケージエディタは出力ファイルを再配置可能なまま残すことができます。
リンクの結果得られたオブジェクトファイルをさらにリンクの対象にすることができるので、リンクを何段階かに分けて行えることから、この機能をインクリメンタルリンクと呼びます。
この機能は、モジュール分割されたソフトウェア開発で、再リンクの手間を少なくするのに非常に有用です。

5.4.2 インクリメンタルリンク時の処理

インクリメンタルリンクでは、入力セクションを結合して出力セクションを作成し、シンボルの解決を行なった結果がリロケータブルオブジェクトファイルに出力されます。ライブラリファイルの検索、メモリ上への配置、相対項の再配置等の処理は行われません。また、シンボルの解決を行なった結果、未定義シンボルが残っていてもエラーとはなりません。
インクリメンタルリンクを行う("-r" オプションを指定してリンクを行う)場合は、以下の点で通常のリンクと異なる指定が必要です。

- コマンド言語ファイルにメモリ定義部を記述すると、無視されます。これは、インクリメンタルリンクではメモリ上への配置は行われないためです。
- セクション命令部では、セクションの結合のみを指定します。出力セクションのメモリ配置を指定した場合はエラーとなります。
- "-o" オプションによる出力ファイルの指定が必要です。これは、"-o" オプションでファイル名を指定しないと、出力ファイルと入力ファイルが同じ名前になってしまうためです。
- "-k"オプションや代入文は使用できません。

5.4.3 インクリメンタルリンクの例

```
testa.rel  testb.rel  testc.rel
MEMORY  {
    data.m : org=0x0100  len=0x0380
    code.m : org=0x8000  len=0x8000
}
SECTIONS  {
    OUT_CODE : { *(code_sec) }
    OUT_DATA : { *(data_sec) }
}
```

例えば、上記のようなコマンド言語ファイルによって作成されるアブソリュートオブジェクトファイルと同じファイルを、インクリメンタルリンクを使って 2 回のリンク作業により作る場合は、コマンド言語ファイルを以下のように記述します。

1回目のリンク時のコマンド言語ファイル

```
-r                              /* インクリメンタルリンク指定 */
-otestrel.rel                   /* 出力ファイル名の指定 */
testa.rel                       /* 入力ファイル名の指定 */
testb.rel
                                /* メモリ定義部は指定できない */
SECTIONS {                      /* セクションの結合のみの指示 */
    R_CODE : { *(code_sec) } /* code セクションを結合 */
    R_DATA : { *(data_sec) } /* data セクションを結合 */
}
```

リロケータブルオブジェクトファイル testa.rel、testb.rel 中の code_sec セクションを結合し R_CODE セクションを、data_sec セクションを結合し R_DATA セクションを作り、testrel.rel ファイルに出力します。

2回目のリンク時のコマンド言語ファイル

```
testc.rel
testrel.rel
MEMORY {     /* メモリ定義部を指定 */
   data : org=0x0100 len = 0x0380
   code : org=0x8000 len = 0x8000
}
SECTIONS {
   OUT_CODE : { *(R_CODE) *(code_sec) }
                              /* CODEセクションをROMに配置 */
   OUT_DATA : { *(R_DATA) *(data_sec) }
                              /* DATAセクションをRAMに配置 */
}
```

1 回目のインクリメンタルリンクで作成したリロケータブルオブジェクトファイル testrel.rel と、デバッグ修正して再アセンブルしたリロケータブルオブジェクトファイル testc.rel をリンクし、アブソリュートオブジェクトファイルを作成します。

上記の 2 回のリンクの結果作成されるファイルは、最初のコマンド言語ファイルにより 1 回のリンクで作成されるファイルとほぼ同じになります。

第 6 章 TX19 リンカ固有仕様

この章では、TX19 リンカ固有の仕様について説明します。なお TX19 固有ではないリンカの仕様については第 1 章～第 5 章を参照して下さい。

6.1 メモリ配置について

6.1.1 配置アドレス指定されたリロケータブルセクションの配置

下表に入力するセクションの種類と配置先のメモリの関係を示します。

配置先メモリ / 入力セクション	DATA			ROMDATA			CODE		ユーザ定義メモリ
	S	M	L	S	M	L	M	L	
DATA.S(mall)	○ 例 1	E205 例 2	E205	○	E205	E205	E205	E205	E205
DATA.M(medium)	E235 E205	○	E235 E205	E235 E205	○	E235 E205	E235 E205	E235 E205	E235 E205
DATA.L(large)	○	○	○	○	○	○	○	○	○
ROMDATA.S	W524	W524 E205	W524 E205	○	E205	E205	E205	E205	E205
ROMDATA.M	W524 E235 E205	W524	W524 E235 E205	E235 E205	○	E235 E205	E235 E205	E235 E205	E235 E205
ROMDATA.L	W524	W524	W524	○	○	○	○	○	○
CODE.M	W524	W524	W524	○	○	○	○	○	○
CODE.L	W524	W524	W524	○	○	○	○	○	○

○ : No Error

W524: CODE or ROMDATA section is allocated in DATA area with 'org'

CODE または ROMDATA タイプの入力セクションをリンクコマンドファイルの org で DATA 領域に該当するアドレスを指定して配置しました。

E205: Can't allocate “<section_name>”

出力セクションが割り当てられませんでした。

E235:medium section (“<section_name>”) must be allocated from “<address 1>” to “<address 2>”

DATA.M または ROMDTA.M 属性を持つセクションは“<address 1>”から “<address 2>” で表される領域に配置されなければなりません。

例 1,2:後続に例を示します。

ROMDATA／CODE セクションを、既定義 DATA メモリへ配置するように配置アドレス指定をした場合ワーニングを出力し、既定義データメモリに配置したまま処理を続行します。

small 属性を持つ DATA／ROMDATA セクションを、small 属性を持つ DATA／ROMDATA メモリ以外へ配置するように配置アドレス指定をした場合、エラーが出力されます。

medium 属性を持つ DATA／ROMDATA セクションを、medium 属性を持つ DATA／ROMDATA メモリ以外へ配置するように配置アドレス指定をした場合、エラーが出力されます。

large 属性を持つ DATA／ROMDATA セクションに配置アドレス指定をした場合指定されたメモリの配置属性・種類に関わらずセクションを配置します。

コードセクションに配置アドレス指定をした場合、指定されたメモリの配置属性・種類に関わらずセクションを配置します。

配置アドレスを指定して入力セクションを配置する例として、下記 small 属性を持つ DATA セクション（セクション名 data_s）を配置する場合を考えます。

アセンブラプログラム（ファイル名: test1.asm）

```
Module test1

Data_s section data small ;セクション data_s の種類は DATA.S
Isym_s1 dsw 1
Isym_s2 dsw 1
```

下記、例 1 のリンクコマンドファイルを利用して、入力セクション data_s (種類:DATA、属性 small)をメモリ DATA.S へ配置するとエラーにはなりません。

例 1　　リンクコマンドファイル

```
Memory {
        DATA.S          :       org=0x00000,    len=0x04000
        DATA.M          :       org=0x10000,    len=0x08000
}
sections
{
    out1    org=0x0     :{test1.rel(data_s)}
/* セクション data_s（種類:DATA、属性 small）を org 指令でアドレス
0x0 へ配置指定。アドレス 0x0 に定義されたメモリは DATA.S 属性を持つ。 */
 }
```

しかし下記、例 2 のリンクコマンドファイルを利用して、入力セクション data_s（種類:DATA、属性 small）をメモリ DATA.M へ配置すると、エラーとなり下記のメッセージが出力されます。

例 2　リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       DATA.M        :      org=0x10000,  len=0x08000
}

sections
{
   out1    org=0x10000 :{test1.rel(data_s)}
 /* セクション data_s（種類:DATA、属性 small）を org 指令で
    アドレス 0x10000 へ配置指定。
    アドレス 0x10000 に定義されたメモリは DATA.M 属性を持つ。 */

}
```

出力されるメッセージ

```
TULINK-Error-205: Can't allocate " out1"
```

6.1.2 開始アドレス指定されたリロケータブルセクションの配置

下表に入力するセクションの種類と配置先のメモリの関係を示します。

配置先メモリ / 入力セクション	DATA			ROMDATA			CODE		ユーザ定義メモリ
	S	M	L	S	M	L	M	L	
DATA.S(mall)	○ 例 1	E205 例 2	E205	W523	W523 E205	W523 E205	W523 E205	W523 E205	E205
DATA.M(medium)	E235 E205	○	E235 E205	W523 E235 E205	W523	W523 E235 E205	W523 E235 E205	W523 E235 E205	E235 E205
DATA.L(large)	○	○	○	W523	W523	W523	W523	W523	○
ROMDATA.S	○	E205	E205	W523	W523 E205	W523 E205	W523 E205	W523 E205	E205
ROMDATA.M	E235 E205	○	E235 E205	W523 E235 E205	W523	W523 E235 E205	W523 E235 E205	W523 E235 E205	E235 E205
ROMDATA.L	○	○	○	W523	W523	W523	W523	W523	○
CODE.M	○	○	○	W523	W523	W523	W523	W523	○
CODE.L	○	○	○	W523	W523	W523	W523	W523	○

○ : No Error

W523:Starting address of CODE or ROMDATA area is specified in ’addr’

リンクコマンドファイルの’addr’開始アドレスが CODE あるいは ROMDATA 領域に該当しています。

E205: Can't allocate “<section_name>”

出力セクションが割り当てられませんでした。

E235:medium section (“<section_name>”) must be allocated from “<address 1>” to “<address 2>”

DATA.M または ROMDTA.M 属性を持つセクションは“<address 1>”から“<address 2>”で表される領域に配置されなければなりません。

例 3,4:後続に例を示します。

入力セクションを、既定義 DATA メモリ以外へ開始アドレス指定した場合、ワーニングを出力し処理を続行します。

small 属性または medium 属性を持つ DATA／ROMDATA セクションを、同じ種類(DATA／ROMDTA)・配置属性(S／M)を持つメモリ以外へ開始アドレスを

指定した場合、エラーが出力されます。

large 属性を持つ DATA／ROMDATA セクションに開始アドレス指定をした場合指定されたメモリの配置属性・種類に関わらずセクションを配置します。

コードセクションに開始アドレス指定をした場合、指定されたメモリの配置属性・種類に関わらずセクションを配置します。

リンクコマンドファイル中のセクション命令部で、入力 DATA セクションに対し、配置アドレスと開始アドレスの両方を指定した場合、入力セクションと配置アドレスで指定されたメモリとの配置属性・種類が一致しなくても、エラーにはなりません。

開始アドレスを指定し配置する例として、small 属性を持つ DATA セクション(セクション名 data_s)を配置する場合を考えます。

アセンブラプログラム（ファイル名: test1.asm）

```
Module test1

Data_s section data small ;セクション data_s の種類は DATA.S
Isym_s1 dsw 1
Isym_s2 dsw 1
```

下記、例 3 のリンクコマンドファイルを利用して、入力セクション data_s (種類:DATA、属性 small)をメモリ DATA.S へ配置するとエラーにはなりません。

例 3　リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       DATA.M        :      org=0x10000,  len=0x08000
}

sections
{
  out1   addr=0x0     :{test1.rel(data_s)}
 /* セクション data_s（種類:DATA、属性 small）を addr 指令で
    アドレス 0x0 へ配置指定。
    アドレス 0x0 に定義されたメモリは DATA.S 属性を持つ。 */
}
```

しかし、次の例 4 のリンクコマンドファイルを利用して、入力セクション data_s (種類:DATA、属性 small)をメモリ DATA.M へ配置すると、エラーとなり下記メッセージが出力されます。

例4　リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       DATA.M        :      org=0x10000,  len=0x08000
}

sections
{
  out1    addr=0x10000 :{test1.rel(data_s)}
 /* セクション data_s（種類:DATA、属性 small）を addr 指令で
    アドレス 0x10000 へ配置指定。
    アドレス 0x10000 に定義されたメモリは DATA.M 属性を持つ。 */

}
```

出力されるメッセージ

```
TULINK-Error-205: Can't allocate " out1"
```

6.1.3 出力メモリが指定されているセクションの配置

下表に入力するセクションの種類と、出力メモリ指定されたメモリとの関係を示します。

配置先メモリ / 入力セクション	DATA			ROMDATA			CODE		ユーザ定義メモリ
	S	M	L	S	M	L	M	L	
DATA.S(mall)	○ 例 1	E226 E227 例 2	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E211
DATA.M(medium)	E226 E227	○	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E235 E211
DATA.L(large)	E226 E227	E226 E227	○	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	○
ROMDATA.S	E226 E227	E226 E227	E226 E227	○	E226 E227	E226 E227	E226 E227	E226 E227	E211
ROMDATA.M	E226 E227	E226 E227	E226 E227	E226 E227	○	E226 E227	E235 E211	E226 E227	E235 E211
ROMDATA.L	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	○	E226 E227	○	○
CODE.M	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	○	○	○
CODE.L	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	E226 E227	○	○	○

○： no error

E235:medium section (“<section_name>”) must be allocated from “<address 1>” to “<address 2>”

DATA.M または ROMDTA.M 属性を持つセクションは“<address 1>”か“<address 2>”で表される領域に配置されなければなりません。

E211: No space for “<section_name>” in “<memory_name>”

出力セクションに割り当てるメモリが残っていません。

E226:Can't allocate “<section_name>” to “<memory_name>”

セクション属性どおりにメモリ配置できません。

E227:Attributes are mismatch between section “<section_name>” and memory

セクションと、それが配置されるメモリの属性が異なります。

例 5,6: 後続に例を示します。

出力メモリ指定したメモリが既定義メモリの場合には、配置するセクションの種類・配置属性と配置先メモリの種類・配置属性が一致していなければ、エラーが出力されます。ただし、コードセクションをコードメモリへ配置する場合は、セクションとメモリの配置属性(M／L)が一致しなくてもエラーにはなりません。

出力メモリ指定したメモリがユーザ定義メモリの場合は、セクションの種類によって必要なメモリ属性が配置先のメモリ領域になければエラーが出力されます。ただし、small または medium 属性を持つ DATA／ROMDATA セクションは、セクションの種類によって必要なメモリ属性（RWXI 属性)が配置先のメモリ領域にあってもエラーが出力されます。

出力メモリ指定し配置する例として、下記 small 属性を持つ DATA セクション(セクション名 data_s)を配置する場合を考えます。

アセンブラプログラム（ファイル名: test1.asm）

```
Module test1

Data_s section data small ;セクション data_s の種類は DATA.S
Isym_s1 dsw 1
Isym_s2 dsw 1
```

下記、例 5 リンクコマンドファイルを利用して、入力セクション data_s（種類:DATA、属性 small）をメモリ DATA.S へ配置するとエラーにはなりません。

例 5　リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       DATA.M        :      org=0x10000,  len=0x08000
}

sections
{
  out1    :{orgtst2.rel(data_s)} > DATA.S
 /* セクション data_s（種類:DATA、属性 small）を
    メモリ DATA.S へ出力メモリ指定。 */

}
```

しかし下記、例 6 のリンクコマンドファイルを利用して、入力セクション data_s（種類:DATA、属性 small）を、メモリ DATA.M へ配置するとエラーとなり下記のエラーメッセージが出力されます。

例 6　リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       DATA.M        :      org=0x10000,  len=0x08000
}

sections
{
  out1   :{orgtst2.rel(data_s)} > DATA.M
 /* セクション data_s（種類:DATA、属性 small）を
    メモリ DATA.M へ出力メモリ指定。 */

}
```

出力されるメッセージ

```
TULINK-Error-227:
 Attributes are mismatch between section "out1" and memory
TULINK-Error-226: Can't allocate "out1" to "DATA.M"
```

6.2 TX19 の既定義メモリ

6.2.1 small 属性を持つ既定義 DATA・ROMDATA メモリ

アドレス 0x0～0x7FFF, 0xFFFF8000～0xFFFFFFFF の領域をスモール領域と呼びます。Small 属性を持つ既定義メモリ DATA.S と ROMDATA.S はスモール領域内に定義されなければなりません。既定義メモリ DATA.S、または ROMDATA.S が　リンクコマンドファイルでスモール領域以外に定義された場合はエラーが出力されます。また、既定義メモリ DATA.S と ROMDATA.S に配置されるセクションを　リンクした結果スモール領域からあふれてしまった場合もエラーが出力されます。

リンクコマンドファイル

```
Memory {
       DATA.S        :      org=0x00000,  len=0x04000
       ROMDATA.S             :      org=0x07000,
len=0x04000
     /* スモール領域以外に定義された場合はエラー */
}
```

6.2.2 medium 属性を持つ既定義 DATA・ROMDATA メモリ

リンクコマンドファイルで定義された medium 属性を持つ既定義メモリ DATA.M と ROMDATA.M の開始アドレスのうち、どちらか小さい方のアドレスから、DATA.M と ROMDATA.M の終了アドレスのうち、どちらか大きい方のアドレスまでの領域を medium 領域と呼びます。この medium 領域の大きさは 64k バイト以内でなければなりません。リンクコマンドファイルで定義された medium 領域の大きさが 64k バイトを超えた場合はエラーが出力されます。また、medium 領域に配置されるセクションの大きさが 64k バイトを超えてしまった場合もエラーが出力されます。

リンクコマンドファイル

```
Memory {
        DATA.M          :       org=0x10000,  len=0x08000
        ROMDATA.M               :       org=0x19000,
len=0x08000
      /* この例の medium 領域は 0x10000～0x20FFF となり
         64k バイトを超えているのでエラーになる。          */
}
```

6.2.3 既定義 CODE メモリ

既定義 CODE メモリの配置属性は、既定義 DATA・ROMDATA メモリのように配置アドレスに対する制限やサイズの制限はありません。また small 属性を持つ CODE メモリ(CODE.S)は定義できません。

CODE セクションは異なる配置属性を持つメモリへ配置してもエラーになりません。例えば CODE.M セクションをメモリ CODE.L へ配置してもエラーにはなりません。

6.3 アセンブラ演算子_ _gp

リンクコマンドファイルのメモリ定義部で定義された medium 領域の開始アドレスをもとに、アセンブラ演算子_ _gp の演算を行います。

アセンブラ演算子_ _gp はリンクコマンドファイルで定義された medium 領域の開始アドレスに値 0x8000 を加えた値と置換されます。

リンクコマンドファイルに既定義メモリ DATA.M と ROMDATA.M の両方とも定義されていない場合、medium 領域は未定義としてアセンブラ演算子_ _gp は値 0x0+0x8000 と置換されます。以下に例を上げて説明します。

アセンブラプログラム

```
Gptst1 section code large
J       _ _gp
```

リンクコマンドファイル

```
Memory {
        DATA.S       : org=0x00000,   len=0x04000
        ROMDATA.M    : org=0x10000,   len=0x08000
        DATA.M       : org=0x18000,   len=0x08000
        usrarea(RX)      :     org=0x0100000,
len=0x01000
```

この例ではリンクコマンドファイルに定義されたメモリ ROMDATA.M の開始アドレスは 0x10000, DATA.M の開始アドレスは 0x18000 です。アセンブラ演算子 _ _gp はこれらの値のどちらか小さい方に値 0x8000 を加えた値に置換されるので、この場合は 0x18000（0x10000＋0x8000）となります。そして、j 命令に対する　コードは下記のようになります。

```
08 00 60 00  （0000 1000 0000 0000 0110 0000 0000 0000 ）
              j 命令   0x18000 を右へ 2bit シフトした値
```

6.4 追加メッセージ

125: Illegal endian in "<filename>"
異なるエンディアンのファイル同士を同一ファイルへリンクできません。

234: Can't allocate medium sections (Small Data Area)
リンクコマンドファイルに DATA.M、または ROMDTA.M 属性を持つメモリが定義されていません。DATA.M、または ROMDTA.M 属性を持つセクションをメモリへ配置するためにはリンクコマンドファイルに DATA.M、または ROMDTA.M 属性を持つメモリの定義が必要となります。

235: Medium section ("<section_name＞") must be allocated from"<address 1> " to "<address 2>"
DATA.M または ROMDTA.M 属性を持つセクションは "<address 1>"から"<address 2>"で表される領域に配置されなければなりません。

236: Medium memory (Small Data Area) too big, maximum 64k bytes.
DATA.M と ROMDATA.M 属性を持つメモリに配置されるセクションはメモリの開始アドレスでアドレスが若い方(アドレスの数値の小さい方)から 64k バイト以内に配置されなければなりません。

526: Section "<section_name>" at "<address>" load value overflow. Truncated
直値を引数として持つ命令において直値が命令の許容範囲に入っていません。上位バイトは切り捨てられます。

第1章 概要

マクロプロセッサの主要な機能は、文字列の置き換えです。

マクロ処理の対象　マクロプロセッサの処理対象はソースファイル中のマクロ処理言語で記述された部分のみです。マクロプロセッサはマクロ処理言語に出会うと、その部分を展開し出力ファイルへ出力します。

マクロ処理の非対象　マクロプロセッサが処理の対象としない部分は、単純に出力ファイルへコピーされます。

第2章 基本規則

2.1 ソースプログラム

ソースプログラムはマクロプロセッサが処理の対象とするマクロ処理言語で記述された部分と、処理の対象としない部分から構成されます。

1行の文字数　ソースプログラムは1行1024文字以内でなければなりません。

2.2 文字セット

マクロ処理部を記述するときには表 2.1 で示した文字が使用できます。
マクロ機能の文字列部とコメントには表 2.1 以外の文字や漢字も使うことができます。

表2.1 マクロ処理部に記述できる文字セット

文字	名称	ASCIIコード
0 - 9	数字	0x30 - 0x39
A - Z	英大文字	0x41 - 0x5a
a - z	英小文字	0x61 - 0x7a
!	感嘆符	0x21
"	ダブルクォーテーション	0x22
#	シャープ	0x23
$	ドルマーク	0x24
%	パーセント	0x25
&	アンパサンド	0x26
'	シングルクォーテーション	0x27
(	左括弧	0x28
)	右括弧	0x29
*	アスタリスク	0x2a
+	プラス	0x2b
,	カンマ	0x2c
-	マイナス	0x2d
.	ピリオド	0x2e
/	スラッシュ	0x2f
:	コロン	0x3a
;	セミコロン	0x3b
<	不等号	0x3c
=	等号	0x3d
>	不等号	0x3e
?	クエッションマーク	0x3f
@	アットマーク	0x40
¥	バックスラッシュ	0x5c
_	アンダースコアー	0x5f
|	縦線	0x7c
~	波線	0x7e
(SP)	スペース	0x20
(TAB)	タブ	0x09

2.3 シンボルの記述方法

シンボルは次の規則によって記述します。

- 記述できる文字は次の3種類です。
  - 英文字(A - Z,a - z)
  - 数字(0 - 9)
  - アンダースコア(_)
- シンボルは、32文字以内の文字列であることが必要です。33文字以上のシンボルが定義された場合、32文字目までが有効となり、それ以降は無視されます。
- 先頭文字は、英文字またはアンダースコア'_'でなければなりません。先頭が数字の場合は、数値とみなされます。
- 2文字目以降は英字,数字,またはアンダースコア'_'で続けます。
- ユーザ定義シンボルでは、大文字と小文字は区別されます。

【例】

```
;正しいシンボル名
MACRO_NAME
_MACRO_NAME
ABCDEFGHIJKLMNOPQRSTUVWXYZ123456 ;32文字のシンボル名です
Symbol_name
symbol_name

;注意が必要なシンボル名
;次の2つのシンボルはどちらも33文字あります。
;シンボル名は32文字までが有効です。
;従って、この2つのシンボルは同じシンボルと解釈されてしまいます。
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567
ABCDEFGHIJKLMNOPQRSTUVWXYZ1234568

;正しくないシンボル名
1SYMBOL   ;シンボル名の先頭に数字は使えません。
SYMBOL(1);使用できない記号がシンボル名に使われています。
```

2.4 予約語

マクロプロセッサによって既に定義されているためユーザが定義できないシンボルです。次のものが予約語です。

マクロ処理機能

グローバルカウンタ

制御命令

マクロ処理機能	ASSIGN	DEFINE	ELSE	EQS	EVAL
	EXIT	FI	GES	GTS	IF
	IN	LEN	LES	LOCAL	LTS
	MACLIB	NES	OUT	REPEAT	SET
	SUBSTR	THEN	TRIGGER	WHILE	
グローバルカウンタ	GC0	GC1	GC2	GC3	GC4
	GC5	GC6	GC7	GC8	GC9
制御文	EJECT	EJ	GEN	GE	GENONLY
	GO	INCLUDE	IC	LIST	LI
	NOLIST	NOLI	TITLE	TT	

グローバルカウンタ　グローバルカウンタは将来の拡張のための予約語です。またグローバルカウンタは処理機能、制御文の予約語と異なりユーザ定義の識別子としても使用できます。

2.5 定数

定数はマクロ機能で式が要求される箇所に記述されます。定数には次の種類があります。

- 数値定数
- 文字定数
- SET機能で定義されたシンボル

2.5.1 数値定数

-Hオプション　数値定数の記述方法には次の2通りがあります。ディフォルトはUNIX記法です。IEEE第2記法を使う場合は、-Hオプションを指定します。

UNIX記法
- 8進数　0(ゼロ)で始る0-7の数字列で表します。
- 10進数　0(ゼロ)以外で始る0-9の数字列で表します。
- 16進数　0x(ゼロ エックス)で始まる0-9,a-f,A-Fの英数字列で表します。

IEEE第2記法
- 2進数　Bまたはbで終了する0,1の数字列で表します。
- 8進数　O(オー),oまたはQ,qで終了する0-7の数字列で表します。
- 10進数　D,dで終了する0-9の数字列または単に0-9の数字列で表します。
- 16進数　0-9で始まり、H,hで終了する0-9,a-f,A-Fの英数字列で表します。

```
UNIX記法     IEEE第2記法
             1B              ; 2進
0765         765q            ; 8進数
246          0246            ; 10進数
345          345             ; 10進数
65535        65535d          ; 10進数
0xffff       0ffffh          ; 16進数
```

2.5.2 文字定数

- 文字定数は'(一重引用符)で囲まれた4文字以内の文字列で、整数定数の代わりに用いることができます。
- 引用符自身を文字定数とする場合は引用符を2個続けて書きます。
- 5文字以上の場合は下位4文字のみが使用されます。

```
ld  a,?eval('A'+2)    ; ld a,0x43
ld  a,?eval('''')     ; ld a,0x27
```

2.5.3 SET機能

SET機能で定義されたシンボルは数値として扱われます。

```
?set(ABC,12)          ; シンボルABCは数値12(10進数)として扱われる
```

2.6 式と演算子

式の値　式の値は文字定数を除いて符号付き32ビット整数として計算されます。計算中にオーバーフロー,アンダーフローが発生した場合には、32ビットより上の部分は無視されます。

演算子　演算子の種類,意味と評価順序は次のとおりです。

表2.2 演算子

演算	演算子	記号	表記
算術演算	単項マイナス	-	-項
	加算	+	項+項
	減算	-	項-項
	乗算	*	項*項
	除算	/	項/項
	剰余算	%	項%項
論理演算	単項否定	~	~項
	論理積	&	項&項
	論理和	|	項|項
	排他的論理和	^	項^項
	右シフト	>>	項>>項
	左シフト	<<	項<<項
関係演算	比較	==	項==項
	比較	!=	項!=項
	比較	>	項>項
	比較	<	項<項
	比較	>=	項>=項
	比較	<=	項<=項

表2.3 演算子の優先順位

優先順位	演算子記号
1	単項演算子(-,^)
2	乗算,除算,剰余算(*,/,%)
3	加算,減算(+,-)
4	シフト演算(>>,<<)
5	比較(<,>,>=,<=)
6	比較(==,!=)
7	論理積(&)
8	排他的論理和(^)
9	論理和(|)

被演算子　被演算子(項)は定数と関数参照です。関数参照は次のマクロ機能の参照です。

LEN	EVAL				
EQS	NES	GTS	GES	LTS	LES

2.7 コメント

コメントは;(セミコロン)で始まります。

ダブルコメント記号

;(セミコロン)を2個続けると、コメントが出力ファイルに出力されません。

ダブルコメント記号を使用しても、DEFINE機能,WHILE機能,REPEAT機能の定義部のコメントは出力されます(展開形には出力されません)。

マクロ定義	返答値

`?repeat(3)(`
`ld  a,b ;;Load from b to a`
`)`
	`; +?repeat(3)(`
	`; +ld  a,b ;;Load from b to a`
	`; +)`
	`ld  a,b`
	`ld  a,b`
	`ld  a,b`

第3章 マクロプロセッサの基本規則

マクロプロセッサの機能は次のとおりです。

種類	名称	機能
定義	TRIGGER	トリガキャラクタの定義
	SET	シンボルの値の定義
	DEFINE,LOCAL	シンボルの定義,ユーザマクロの定義
	ASSIGN	二つのシンボルへの文字列の割当て
条件判断	IF,THEN,ELSE,FI	条件分岐
	EXIT	マクロ展開の打切り
繰返し	REPEAT	マクロ展開の繰返し(回数)
	WHILE	マクロ展開の繰返し(条件)
マクロコール	MACLIB	マクロライブラリファイルからのマクロの取込み
	マクロ参照	ユーザ定義マクロ参照
補助	SUBSTR	文字列の取出し
	EQS,NES,GTS, GES,LTS,LES	文字列の比較
	LEN	文字列の長さの計算
	EVAL	式の計算
	IN	コンソールからの入力
	OUT	コンソールへの出力
	エスケープ機能	文字列をそのまま出力
	ブラケット機能	文字列をそのまま出力

3.1 文字列処理の基本

トリガキャラクタ

- マクロ処理の指示を与える記号をトリガキャラクタと呼びます。
  トリガキャラクタ(マクロ機能)の使用位置には注意が必要です。
  トリガキャラクタはTRIGGER機能により変更できます。
  本書では、トリガキャラクタは特に示さない限り?を使用します。
- マクロ中の文字列(テキスト)は左右を括弧で囲みます。
  以下の例では、idがmacroidに置換されます。

```
?define(id)(macroid)
```

- 左右の括弧囲みの中はデリミタ(区切り記号)である,(コンマ)を除き、スペース,水平タブを含め全て文字列の一部とみなします。
  以下の例では、idがmac idに置換されます。

```
?define(id)(mac id)       ; id=mac id
```

バランスしたテキスト

- 文字列中の括弧はバランスしていなければなりません。つまり、左括弧はそれに対応する右括弧を持ち、かつ各右括弧はそれに先行し対応する左括弧を持たなければなりません。

```
This (is) a (balanced (text))       ; バランスしたテキスト
This is (not) (balanced (text)      ; バランスしたテキストでない
```

- 文字列中で括弧の左右のバランスを崩す時、またはデリミタを文字列の一部として使用する場合はエスケープ機能を使います。
  以下の例では、"?1"に続く"("は マクロ処理されません。

```
This is (not) ?1(balanced (text)    ; バランスしたテキストとなる
```

ヌルストリング

- ()(括弧)だけの文字列は空(長さ0)の文字列となります。同様に,"("または ",,"または,")"で表された文字列も空の文字列となります。
  以下の例では、idがnullコードに置換されます。

```
?define(id)()             ; id=null
```

マクロのネスト　❑　マクロ機能は最大8レベルまでネスト可能です。

次の機能はネストレベルをカウントします。

DEFINE機能　ASSIGN機能　IF機能

REPEAT機能　WHILE機能　ユーザ定義マクロ参照

```
?define(enzan(operate))(                    ; ネストレベル1
?if(?eqs(add,?operate)) then (              ; ネストレベル2
        adc a,b
) else (
     ?if(?eqs(sub,?operate)) then (         ; ネストレベル3
            sbc  a,b
       ) else (
          ld  a,b
          ?if(?eqs(mul,?operate)) then (    ; ネストレベル4
                      mul  hl,a
                ) else (
                      div  hl,a
            )fi
     )fi
 )fi
)
                                            返答値
?enzan(add)                    =>           adc a,b
?enzan(sub)                    =>           sbc a,b
?enzan(mul)                    =>           ld a,b
                                            mul hl,a
?enzan(div)                    =>           ld a,b
                                            div hl,a
```

3.2 マクロ機能の返答値

返答値とは　マクロ処理では、マクロの返答値(出力ストリーム)が他のマクロの変数(入力ストリーム)となることがあります。最初のマクロ処理では入力ストリームはソースファイルです。また、最後のマクロ処理では出力ストリームは出力ファイルになります。マクロプロセッサは入力ストリームを一定の規則にしたがって出力ストリームに写像(変換)します。この写像を返答値と呼びます。マクロ機能は返答値が空(長さ0の文字列)のグループと返答値があるグループに分けられます。

返答値が空のグループ

- マクロ機能(?)が行の最初になければなりません(空白,水平タブは挿入可能)。
  返答値のない機能は次のとおりです。
  TRIGGER　SET　DEFINE　ASSIGN　EXIT
  REPEAT　WHILE　MACLIB　OUT
  IF機能のテキスト部以外の記述

```
?set(abc,0x123)
abcの値は0x123である。?set(abc,0x123)自身は返答値がない。
```

返答値があるグループ

- ユーザ定義マクロの参照と関数機能には返答値があります。
- これらの返答値は文字列です。
- ソースプログラム中どの位置にでも書けます。
- SUBSTR機能の場合は文字列が分割されたものが返答値となります。
- その他の機能については評価後の値の16進数表示の文字列となります。
  返答値のある機能は次のとおりです。

```
SUBSTR　　IN
LEN　EQS　NES　GTS　GES　LTS　LES
エスケープ機能　　ブラケット機能
IF機能のテキスト部
ユーザ定義マクロ参照
```

```
?substr(abcdef,2,2)  ; ?substr(abcdef,2,2)自体が文字列bcを返す
?len(abc)            ; ?len(abc)自体が文字列0x03を返す
```

3.3 返答値と出力ファイル編集

返答値が空のグループ

このグループは返答値が空、つまり出力ファイル(ストリーム)に渡すものがありません。従って、GEN 制御命令指定時には、ユーザの便宜のために、空の返答値を返すマクロ機能をコメント行の形式で出力します。（GEN 制御命令については「第 5 章 制御命令」を参照して下さい。）

```
?set(abc,0x123)            ; 返答値が無いため、何も出力されない
?gen
?set(abc,0x123)            ; 出力は ;+set(abc,0x123)
ld a,b  ?set(abc,0x123)    ; エラー
```

返答値があるグループ

返答値はマクロ処理の成果です。これらはマクロ処理の記述を返答値によって置換した結果が出力されます。

```
?substr(abcdef,2,2)   ; 出力ファイルに文字列bcが返される
?len(abc)             ; 出力ファイルに文字列0x03が返される
ld a,?len(abc)        ; 出力は ld a,0x03 となる
```

3.4 テキストの有効範囲

マクロプロセッサはソースプログラムを処理する場合、ある一定範囲内の文字列のみを処理します。マクロプロセッサによって処理される範囲をテキストの有効範囲と呼びます。テキストの有効範囲は次のとおりです。

- テキスト(ボディ)の開始を示す左括弧のある行は、テキスト部に含めません。
- 左括弧の右側に空白,コメント記号以外の文字があるとエラーです。
- 唯一の例外はDEFINE機能による文字列の定義です。

```
左括弧以降にテキストを記述できないもの(…の部分には何も書けません)
THEN(… または     IF(式) THEN(…
ELSE(… または     )ELSE(…
WHILE(式)(…
REPEAT(式)(…
DEFINE(シンボル(-))(…       ; ここにはLOCAL文も書けません
?repeat(5)  (ld a,b         ; エラー。REPEAT機能は無視されます
             ld a,c         ; 通常のステートメントとみなされます
)                           ; 通常のステートメントとみなされます
```

- テキストの終了を示す右括弧以降にテキストは書けません。
- 右括弧の右側に空白,コメント記号以外の文字があるとエラーです。

```
右括弧以降にテキストを記述できないもの(…の部分に何も書けません)
DEFINE(シンボル)(文字列)…
IF(式)…
)FI…
)…
EXIT[(式)]…
ASSIGN(-)(-)…
SET(-,-)…
OUT(-)…
```

第4章 マクロプロセッサ機能の詳細

4.1 定義機能

TRIGGER機能

【記述形式】 ?TRIGGER(<文字>)

【機能】 新しいトリガキャラクタを定義します。

【規則】
- ❑ トリガキャラクタの初期値は?です。
- ❑ この機能は一行で書いて下さい。
- ❑ 次の文字はトリガキャラクタとして使用できません。

  スペース
  水平タブ
  , (コンマ)
  ' (引用符)
  ( ) + - * / (演算子に使用されている特殊記号)
  ; (コメント記号)
  コントロール文字(20H以下のASCIIコード値を持つ無印字記号)
  シンボル構成文字(A-Z,a-z,0-9,_)
- ❑ トリガキャラクタはソースプログラムで通常使用しない文字を使用します。

【例】
```
?trigger(@)          ;トリガキャラクタを@に変更します
@set(abc,0x123)
```

SET機能

【記述形式】 ?SET(<シンボル>,<式>)

【機能】 シンボルに式の値を定義します。

【規則】
- SET機能で定義されたシンボルはSET機能,DEFINE機能で再定義できます。
- この機能は一行で書いて下さい。

【例】

マクロ定義	返答値
`?set(base,0)`	`null`
`?set(offset,0x20)`	`null`
`ld a,?base+?offset`	`ld a,0x0+0x20`
`ld b,(?base)`	`ld b,(0x0)`
`?set(base,?base+?offset)`	`null`
`?set(offset,?base+?offset*2)`	`null`
`ld a,?base+?offset`	`ld a,0x20+0x60`
`ld b,(?base)`	`ld b,(0x20)`

DEFINE機能

DEFINE 機能には 2 つの機能があり、記述方法も異なります。

本書では、記法 1,記法 2 と呼んでいます。

【記法1】

【記述形式】 ?DEFINE(<シンボル>)(<文字列>)

【機能】 シンボルに文字列を定義します。

【規則】
- ❑ 文字列中の括弧はバランスしていなければなりません。
- ❑ この機能は一行で書いて下さい。

【例】
```
?define(dc)(dog and cat)     ; スペースも含まれる
db   "?dc"                   ; マクロコール
db   "dog and cat"           ; 返答値
```

【記法2】

【記述形式】

```
?DEFINE(<シンボル>[(<仮引数リスト>)])
    [LOCAL <ローカルシンボルリスト>](
    <マクロボディ>
)
```

【機能】 ソースステートメントをブロックとして扱えようにマクロ定義します。

【規則】

- シンボルにはマクロ定義の名前(マクロ名)を記述します。
- 仮引数リストはマクロコール時にマクロボディに渡される仮引数を(コンマ)で区切って複数指定することができます。
- ローカルシンボルにはマクロボディ中に記述するラベル名を指定します。ローカルシンボルはマクロ展開時にラベル名に3桁の数字が付いたシンボルに置き換えられます。したがって、ローカルシンボルにはシンボルの有効長-3以内の長さのシンボルを指定してください。
- ローカルシンボルリストにはシンボルをスペース、水平タブで区切って複数指定することができます。また、ローカルシンボルは2行以上に渡って書くことができます。
- ローカルシンボルの有効範囲はネストされていない一番外側のマクロボディのみです。マクロボディ中の条件判断(IF)や繰返し(REPEAT,WHILE)の中では、ローカルシンボルを使用することはできず、エラーとなります(下記の例)。
- マクロボディには、ソースステートメントを複数記述します。ソースステートメントはバランスしたテキストでなければなりません。
- マクロボディ中でローカルシンボルを使用する場合には、シンボル名の前にトリガキャラクタ(?)を付けて指定します。
- マクロボディ中で他のマクロの定義を行うことも、そのマクロを呼び出す(マクロコール)こともできます。(マクロのネスト)。
- DEFINE機能で定義されたマクロは再定義できません。

【例】

```
?define(ABC(a1))              ; マクロ定義
local l1 l2 (
          ?if(?a1==1) then(
               ld   xix, ?l1  ; if機能のボディ内では使用できずエラー
          )fi
               ld   xiz, ?l2  ;  1番外側のネストではOK
)
```

マクロコール

【記述形式】 ?マクロ名[(<実引数リスト>)]

【機能】 DEFINE機能で定義されたマクロをコール(参照)します。

【規則】
- ❑ マクロコールはマクロを展開したい位置にマクロ名を記述して指定します。
- ❑ 実引数リストはマクロ展開時に渡される実引数で、複数の場合は,(コンマ)で区切ります。

【例-1】 以下は引数もローカル変数もなしのマクロ定義とマクロコールの例です。

```
?define(movedata)(          ; マクロ定義
    ld   hl,source
    ld   de,destination
    ld   bc,count
    lddr
)
?movedata                   ; マクロコール
    ld   hl,source          ; 返答値
    ld   de,destination
    ld   bc,count
    lddr
```

【例-2】 以下は引数ありローカル変数なしのマクロ定義とマクロコールの例です。

```
?define(movedata(source,destination,count))(  ; マクロ定義
    ld   hl,?source
    ld   de,?destination
    ld   bc,?count
    lddr
)
?movedata(16,8,8)           ; マクロコール
    ld   hl,16              ; 返答値
    ld   de,8
    ld   bc,8
    lddr
```

【例-3】　以下は引数もローカル変数もあるマクロ定義とマクロコールの例です。

```
?define(movedata(source,destination,count))     ; マクロ定義
local     typei endmove(                        ; ローカル変数
     cp   a,1
     jp   z,?typei
     ld   hl,?source
     ld   de,?destination
     ld   bc,?count
     lddr
     jp   ?endmove
?typei:   ldir
?endmove
)
?movedata(16,8,8)                               ; マクロコール
               cp   a,1                         ; 返答値
               jp   z,typei000
               ld   hl,16
               ld   de,8
               ld   bc,8
               lddr
               jp   endmove000
typei000:      ldir
endmove000:
```

【例-4】　以下の例は マクロ定義の中にマクロコールがある例です。

```
        ?define(pushxx)(                                ; マクロ定義
        push f
        push wa
        push bc
        push de
        push hl
)
?define(movedata(source,destination,count))     ; マクロ定義
local   typei endmove(
        ?pushxx                                 ; マクロコール
        cp   a,1
        jp   z,?typei
        ld   hl,?source
        ld   de,?destination
        ld   bc,?count
        lddr
        jp   ?endmove
?typei:  ldir
?endmove:
)
?movedata(16,8,8)               ; マクロコール
              push f            ; 返答値(PUSHEXXの返答値でもある)
              push wa
              push bc
              push de
              push hl
              cp   a,1
              jp   z,typei000
              ld   hl,16
              ld   de,8
              ld   bc,8
              lddr
              jp   endmove000
typei000:     ldir
endmove000:
```

【例-5】 以下では、マクロコールの際に、他で定義されたマクロ名a_dataを引数として指定しています。

```
?define(movedata(source,destination,count,add_data));マクロ定義
local    typei endmove(
     ?pushxx
     cp   a,1
     jp   z,?typei
     ld   hl,?source
     ld   de,?destination
     ld   bc,?count+?len(?add_data)
     lddr
     jp   ?endmove
?typei:  ldir
?endmove:
)

?define(a_data)(abcd)

?movedata(16,8,8,?a_data)    ; マクロコール(マクロを引数としている)
              push f         ; 返答値(PUSHEXXの返答値でもある)
              push wa
              push bc
              push de
              push hl
              cp   a,1
              jp   z,typei000
              ld   hl,16
              ld   de,8
              ld   bc,8+0x04     ; 0x04=?len(?a_data)の返答値
              lddr
              jp   endmove000
typei000:     ldir
endmove000:
```

ASSIGN機能

【記述形式】 ?ASSIGN(<シンボル1>,<シンボル2>)(<文字列>)

【機能】 文字列をシンボル1,シンボル2に分割して割当てます。

【規則】
- 文字列中の最初のコンマで文字列を前半と後半に分割し、シンボル1には文字列の前半を、シンボル2には後半を割当てます。
- 文字列が空の時はシンボル1,シンボル2ともにNull stringが割当てられます。
- コンマが文字列中にない場合は文字列全体がシンボル1に割当てられ、シンボル2にはNull stringが割り当てられます。
- 文字列中の括弧はバランスしていなければなりません。
- 定義されたシンボルはASSIGN機能,DEFINE機能で再定義できます。
- この機能は一行で書いて下さい。

漢字の使用
- -Jオプションを使用しない場合は、漢字1文字は文字2個分となります。
- -Jオプションを使用した場合は、漢字1文字が1文字として計算されます。

【例】 マクロ定義

```
?assign(next,slist)(0x10,0x20,0x30)
?while(?len(?next) != 0)(
     ld   a,(?next)
     add  a,0x22
     ld   (?next),b
?assign(next,slist)(?slist)
)
```

返答値

```
     ld   a,(0x10)      ;next=0x10         slist=0x20,0x30
     add  a,0x22
     ld   (0x10),b
     ld   a,(0x20)      ;next=0x20         slist=0x30
     add  a,0x22
     ld   (0x20),b
     ld   a,(0x30)      ;next=0x30         slist=Null
     add  a,0x22
     ld   (0x30),b
```

4.2 条件判断機能

IF機能

【記述形式】

```
?IF(<式>) THEN (
    <テキスト1>
[ ) ELSE (
    <テキスト2>]
)FI

または

?IF(<式>)
THEN (
    <テキスト1>
[ )
ELSE (
    <テキスト2>]
)FI
```

【機能】 式の評価結果が真ならばテキスト1が、偽ならテキスト2を展開します。

【規則】
- ❑ 式の評価結果が0以外であれば結果を真とします。
- ❑ 式の評価結果が0であれば結果を偽とします。
- ❑ ELSE部は不要なら省略できます。
- ❑ 評価結果が偽でELSE部が無い場合は何も展開されません。
- ❑ FIでIF機能の終わりを宣言します。
- ❑ IFとFIは対になっていなければなりません。
- ❑ 式は省略できません。
- ❑ テキスト(テキスト1,テキスト2)に含まれる左右の括弧はバランスしていなければなりません。

【例-1】

```
?if(?eqs(add,?operate)) then (
     adc  a,b            ; 真ならここが出力される
) else (
     SBC  a,b            ; 偽ならここが出力される
)fi
```

【例-2】　以下は、if機能の4レベルのネストを示す例です。

```
?define(enzan(operate))(                    ; ネストレベル1
?if(?eqs(add,?operate))then(                ; ネストレベル2
         adc  a,b
)else(
   ?if(?eqs(sub,?operate)) then (           ; ネストレベル3
        sbc   a,b
    ) else (
         ld   a,b
         ?if(?eqs(mul,?operate)) then (     ; ネストレベル4
              mul  hl,a
              ) else (
              div  hl,a
         )fi
    )fi
)fi
)
```

```
                                    返答値
?enzan(add)                 =>   adc  a,b
?enzan(sub)                 =>   sbc  a,b
?enzan(mul)                 =>   ld   a,b
                                 mul  hl,a
```

EXIT機能

【記述形式】 ?EXIT[(<式>)]

【機能】 式の評価結果が真であれば直接この機能を含むREPEAT機能,WHILE機能,マクロコールによるマクロ展開を打切ります。

【規則】
- 結果が偽であればなにも行いません。
- 式を省略した時は結果を真とします。
- この機能は一行に書いて下さい。

【例】 マクロ定義

```
?set(val,0)
?define(bbb) (
?exit(?val > 2)
?if(?val == 2) then (
     ?1;Last Generation
) else (
     ld a,?eval(?val) ?1;next-?eval(?val)
)fi
?set(val,?val+1)
)
```

マクロコール

```
?bbb
?bbb
?bbb
```

返答値

```
ld a,0x00 ;next-0x00    ; 第1番目のマクロコール
ld a,0x01 ;next-0x01    ; 第2番目のマクロコール
; Last Generation       ; 第3番目のマクロコール
                        ; 第4番目のマクロコールではEXITが実行される
```

4.3 繰返し機能

REPEAT機能

【記述形式】

```
?REPEAT(<式>) (
    <テキスト>
)
```

【機能】 式の値の回数だけテキストを展開します。

【規則】
- ❑ 繰り返し回数は最大255回です。
- ❑ 式の値が0の時はテキストの展開は行われません。
- ❑ 式は省略できません。
- ❑ テキスト部にはEXIT機能を使用できます。
- ❑ テキスト中で使用される括弧は左右がバランスしていなければなりません。

【例-1】 マクロ定義

```
?repeat(4) (
NOP
)
```

返答値

```
NOP
NOP
NOP
NOP
```

【例-2】
```
?DEFINE(BUNK1)                  ; ラベルを自動発番するマクロ定義
LOCAL LAB(
        LD      A,1
        B       ?LAB
?LAB
)
?REPEAT(3)(
?BUNK1                          ; マクロコール
)
   ---返却値---
          LD        A,1         ; 1回目のコール
          B         LAB000
LAB000:
          LD        A,1         ; 2回目のコール
          B         LAB001
LAB001:
          LD        A,1         ; 3回目のコール
          B         LAB002
LAB002:
```

WHILE機能

【記述形式】

```
?WHILE(<式>) (
    <テキスト>
)
```

【機能】　式の評価結果が真の間、テキストを展開します。

【規則】
- 式の評価結果が0以外であれば結果を真とします。
- 式の評価結果が0であれば結果を偽とします。
- 式は省略できません。
- テキスト部にはEXIT機能が使用できます。
- テキスト中で使用される括弧は左右がバランスしていなければなりません。

【例】　マクロ定義

```
?set(count,4)
?while(?count > 0) (
     rr   a
     ?set(count,?count-1)
)
```

返答値

```
rr   a
rr   a
rr   a
rr   a
```

【例】

```
?set(count,1000)              ; これなら1000回展開する
?while(?count > 0) (
     rr   a
     ?set(count,?count-1)
)
```

4.4 マクロコール

MACLIB機能

【記述形式】

?MACLIB <<ファイル名>>
?MACLIB "<ファイル名>"

【機能】 マクロライブラリファイルの中を探して、マクロ定義を取り込みます。そのため、マクロ定義は あらかじめマクロライブラリファイルに書いておく必要があります。MACLIB機能は このマクロライブラリファイルを指定するものです。

【規則】

- マクロコール時に未定義のマクロコールがあると、指定されたマクロライブラリファイルの中を探して、該当するマクロ定義を取り込みます。
- マクロライブラリファイルの検索順序は制御文INCLUDE(ページ191参照)と同一です。
- マクロライブラリ中でも他のマクロを参照するとができます。
- 指定できるファイルは最大8個です。
- マクロライブラリ中に該当するマクロが無ければ未定義マクロとなります。
- この機能は一行で書いて下さい。

【例】 マクロライブラリファイル(mac.lib)

```
?define(abc) (
     ld   hl,(bc)
     div  hl,a
     ld   (bc),hl
)
```

マクロコール

```
?maclib "c:¥my¥mac.lib"          ; ファイルMAC.LIBのマクロ
?abc                             ; マクロabcの参照
```

返答値

```
ld   hl,(bc)                     ; マクロabcの展開
div  hl,a
ld   (bc),hl
```

4.5 補助機能

SUBSTR機能

【記述形式】 ?SUBSTR(<文字列>,<式1>,<式2>)

【機能】 文字列中から式1の値で示されるカラムから、式2の値の数の文字を取出します。

抽出規則
(1) 文字列長 < 式1の値 の場合、取り出される文字列は空となります。
(2) (文字列長 - 式1の値) < 式2の値 の場合、式1で示されるコラムから後の文字が取り出されます。
(3) 式1の値=0 または 式2の値=0 の場合は、取り出される文字列は空となります。
(4) 上記以外の場合、 式1で示されるカラムから式2で示される数の文字が取り出されます。

【規則】
- この機能はプログラム中の文字列が書ける所であればどこにでも書けます。
- この機能は一行で書いて下さい。

漢字の使用
- -Jオプションを使用しない場合は、漢字1文字は文字2個分となります。
- -Jオプションを使用した場合は、漢字1文字が1文字として計算します。

【例-1】

```
マクロ定義                  =>   返答値
?substr(abcdefg,8,1)    =>   null
?substr(abcdefg,3,0)    =>   null
?substr(abcdefg,5,1)    =>   e
?substr(abcdefg,5,100)  =>   efg
```

【例-2】 ローカルシンボルからマクロ展開番号を抽出します。

```
?define(ABC)
 local ln(
       ld xix,?eval(?substr(?ln,3,3)+1)
)
                                        返答値
?ABC   /*1回目のマクロコール*/ =>   ld   xix,0x1
?ABC   /*2回目のマクロコール*/ =>   ld   xix,0x2
?ABC   /*3回目のマクロコール*/ =>   ld   xix,0x3
```

マクロ演算子(EQS,NES,LTS,LES,GTS,GES機能)

【記述形式】

```
?EQS(<文字列1>,<文字列2>)
?NES(<文字列1>,<文字列2>)
?LTS(<文字列1>,<文字列2>)
?LES(<文字列1>,<文字列2>)
?GTS(<文字列1>,<文字列2>)
?GES(<文字列1>,<文字列2>)
```

【機能】 2つの文字列を比較します。

【規則】

- 比較値は文字列のASCII値です。
- 真ならば文字列0xffffffffを、偽ならば0x00を返します。

  ?EQS(文字列1,文字列2) ; 文字列1 == 文字列2 ならば真
  ?NES(文字列1,文字列2) ; 文字列1 != 文字列2 ならば真
  ?LTS(文字列1,文字列2) ; 文字列1 < 文字列2 ならば真
  ?LES(文字列1,文字列2) ; 文字列1 <= 文字列2 ならば真
  ?GTS(文字列1,文字列2) ; 文字列1 > 文字列2 ならば真
  ?GES(文字列1,文字列2) ; 文字列1 >= 文字列2 ならば真
- プログラム中の被演算子として使用できます。
- マクロ演算子は一行で書いて下さい。

【例】

マクロ定義	返答値
`?eqs(abc, abc)`	`0x00` ;スペースも比較対象
`?nes(ABC,AbC)`	`0xffffffff`
`?lts(ABC,abc)`	`0xffffffff`
`?les(ABCD,ABC)`	`0x00`
`?gts(0x11,0x13)`	`0x00`
`?ges(abcde,abcde)`	`0xffffffff`

LEN機能

【記述形式】 ?LEN(<文字列>)

【機能】 文字列の長さをバイト長単位で計算します。

【規則】
- ❑ 長さは数値定数の形式(ASCII文字列)で返されます。
- ❑ スペース,水平タブも一文字としてカウントします。
- ❑ プログラム中の被演算子として使用できます。
- ❑ この機能は一行で書いて下さい。

漢字の使用
- ❑ -Jオプションを使用しない場合は、漢字1文字は文字2個分となります。
- ❑ -Jオプションを使用した場合は、漢字1文字が1文字として計算されます。

【例】

マクロ定義	返答値
`?len(a,B,C)`	`0x05`
`?define(high)(low)`	
`?define(big)(small)`	
`?len(?high ?big)`	`0x09 ;high=3文字,スペース=1文字,BIG=5文字`
`ld a,?len(?high ?big)`	`ld a,09H`

EVAL機能

【記述形式】 ?EVAL(<式>)

【機能】 式を評価します。

【規則】
- 結果は数値定数の形式(ASCII文字列)で返されます。
- プログラム中の被演算子として使用できます。
- この機能は一行で書いて下さい。

【例】

マクロ定義	返答値
`ld a,?eval(2+2)`	`ld a,0x04`
`count equ ?eval(0x11+0x15+0xf20)`	`count equ 0xf46`
`?set(sum1,22)`	
`?set(sum2,0x12)`	
`and hl,?eval(?sum1 <= ?sum2)`	`and hl,0x00`

IN機能

【記述形式】 ?IN

【機能】 コンソールに">"を出力し、キー入力待ちになります。

【規則】
- ❑ この機能は文字列が書けるところであればどこにでも書けます。
- ❑ この機能は一行で書いて下さい。

【補足】
- ❑ 入力された文字列は単純文字列と見なします。したがって、入力文字列中にマクロ機能が使用された場合には、入力時にはエラーとなりませんが、使用方法によっては予期せぬところで展開されることもあります。

【例】
```
address equ ?in          ; ソースプログラム
>0x20                    ; キー入力(0x20)
address equ 0x20         ; 出力ストリーム(ファイル)
```

OUT機能

【記述形式】 ?OUT(<文字列>)

【機能】 コンソールに文字列を出力します。

【規則】
- 文字列出力の復帰改行はTUMPLが行います。
- 文字列は出力の前にTUMPLによってスキャンされ、マクロ機能の参照がある場合はマクロの展開後に出力されます。
- この機能は一行で書いて下さい。

【例】 マクロ定義

```
?out(Type type of processor in system)
?set(ptype,?in)
?out(Type start address in system)
address  equ  ?in
?out(Type SIO baud rate?)
?set(baud,?in)
```

次の様にコンソールへ表示されます。

```
Type type of processor in system
>test                          ; ユーザがtestと入力した場合
Type start address in system
>0x20                          ; ユーザが0x20と入力した場合
Type SIO baud rate?
>9600                          ; ユーザが9600と入力した場合
```

これは次のようにソースプログラムに書いたの同じです。

```
?set(ptype,test)
address  equ  0x20
?set(baud,9600)
```

エスケープ機能

【記述形式】 ?n<文字列>

【機能】 ?nに続くn文字をマクロ処理することなく出力ストリームへ渡します。

【規則】
- ❑ nは1桁の10進数です。
- ❑ スペース,水平タブも1文字として数えます。
- ❑ この機能の中のブラケット機能は実行されます。
- ❑ この機能は一行で書いて下さい。

漢字の使用
- ❑ -Jオプションを使用しない場合は、漢字1文字は文字2個分となります。
- ❑ -Jオプションを使用した場合は、漢字1文字が1文字として計算します。

【例】

マクロ定義	返答値
`evaluate 10?1? of sum`	`evaluate 10? of sum`
`escape?11character`	`escape1character`
`?maccall(January22?1,1993)`	`January22,1993;1`つの実引数となる

ブラケット機能

【記述形式】 ?(<文字列>)

【機能】 ( )内の文字列をマクロ処理することなく出力ストリームへ渡します。

【規則】
- ❑ この機能の中のエスケープ機能は実行されます。
- ❑ この機能は一行で書いて下さい。

【例】 マクロ定義

```
?define(DDATA(llist,label))(
?label:dw     ?llist
)
```

マクロコール

```
?DDATA(?(0x245,0x1da,0x333),tel)
```

マクロ展開後

```
tel:dw    0x245H,0x1da,0x333
```

マクロコール

```
?maccall(one_parameter?(,,?1(,,) ; 実引数 = one_parameter,,(,,
```

第5章 制御命令

制御命令はリスト出力などの制御を行います。
制御命令はソースプログラム中に記述します。
通常のマクロ機能の様にトリガキャラクタを付けて使用します。

EJECT

【記述形式】 ?EJECT

【機能】 リストファイルを改ページします。

【省略形】 ?EJ

【ディフォルト】 なし

【規則】
- ❑ 本命令の後で、改ページを行います。
- ❑ 起動時オプションで-lを指定していない場合、または本命令が記述された場所がnolist制御命令の範囲内の場合は、本命令は無効となります。

【例】
```
?SET(Macro,2)
?EJECT          ; ここで改ページされる
LD  DE,?Macro   ; 新しいページに出力される
```

LIST / NOLIST

【記述形式】

```
?LIST
?NOLIST
```

【機能】 アセンブルリストファイルを出力/非出力します。

?LIST　　アセンブルリストが出力されます。

?NOLIST　アセンブルリストが出力されません。

【省略形】 ?LI

?NOLI

【ディフォルト】 ?LIST

【規則】

- 本命令は「-l」オプションが指定されていない場合は、無視されます。
- ?LIST

  本命令以降NOLIST命令が指定されるまで、リストファイルにリストを出力します。
- ?NOLIST

  本命令以降LIST命令が指定されるまで、リストファイルにリストを出力しません。ただし、NOLISTの範囲内にエラーが発生した場合は、エラー行とエラーメッセージをリストファイルに出力します。

【例】

```
?NOLIST
:     この部分のリストは出力されません
:
?LIST
:     この部分のリストは出力されます
:
?NOLIST
```

TITLE

【記述形式】 ?TITLE "<文字列>"

【機能】 <文字列>をタイトルとしてリストファイルに出力します。

【省略形】 ?TT "<文字列>"

【デフォルト】 ?TITLE "リストファイル名"

【規則】
- 本命令が現れた時点で改ページを行い、次のページの先頭にタイトルを出力します。
- <文字列>は60文字まで指定可能です。60文字を超えた場合は、60文字以降は無視されます。
- <文字列>は印字可能なASCII文字です。
- 起動時オプションで「-l」を指定していない場合、または本命令が記述された場所がnolist制御命令の範囲内の場合は、本命令は無効となります。

【例】 以下の例では、「TEST PROGRAM」という文字列をリストファイルのタイトルに指定しています。

```
?TITLE "TEST PROGRAM"
```

GENONLY / GEN

【記述形式】

?GENONLY

?GEN

【機能】 マクロ定義,マクロコールのリストへの出力の可否を選択します。

【省略形】 ?GO

?GE

【デフォルト】 ?GENONLY

【規則】

- ?GENONLY

  マクロ処理の展開形のみが出力されます。次のテキストは出力されません。

  - 制御文
  - マクロ処理言語での各種定義および返答値のないマクロ機能
  - IF文により選択されなかったソーステキスト

  GENONLY制御命令はEJECT,LIST,NOLIST,TITLE制御文より優先度が高く、GENONLY区間に有るこれらの制御はリストファイル編集には反映されません。

- ?GEN

  マクロ定義,マクロコール,マクロコールの展開を出力ファイルに出力します。

- エラーがあると常にソーステキストとエラーメッセージが出力されます。

  そして、マクロ処理の展開形以外の部分は;+に付けて出力します。

表5.1 GEN/GENONLYとマクロ機能の関係

	GEN	GENONLY
制御文	Y (出力される)	N (出力されない)
TRIGGER機能	Y	N
SET機能	Y	N
ASSIGN機能	Y	N
EXIT機能	Y	N
MACLIB機能	Y	N
OUT機能	Y	N
DEFINE機能	Y	N
REPEAT機能	Y	Y (展開形のみ)
WHILE機能	Y	Y (展開形のみ)
IF機能	Y	N
IF機能のボディ部(真区間)	Y	Y
IF機能のボディ部(偽区間)	Y	N
マクロコール	Y	Y
返答値を持つマクロの参照	Y	Y

【例】

マクロ定義

```
%repeat(3)(
     db 0x03
)
```

出力形式(GEN指定)

```
;+%repeat(3)(
;+ db 0x03
;+)
db 0x03
db 0x03
db 0x03
```

出力形式(GENONLY指定)

```
db 0x03
db 0x03
db 0x03
```

INCLUDE

【記述形式】
```
?INCLUDE <<ファイル名>>
?INCLUDE "<ファイル名>"
```

【機能】 <ファイル名>で指定したソースファイルを読み込みます。

【省略形】 ?IC<<ファイル名>>
?IC"<ファイル名>"

【デフォルト】 なし

【規則】
- ファイル名で指定したファイルを?include命令の位置に読み込みマクロ処理を行います。
- ?include命令は、8段階までネストすることができます。これを超えるネストを行うと、エラーとなります。
- ファイル名を絶対パスで指定した場合は、そのファイルがそのまま本命令の位置に挿入されます。ファイルが存在しなかった場合は、エラーとなります。
- ファイル名を相対パスで指定した場合は、?include<<ファイル名>>の場合と?include"<ファイル名>"の場合では検索の順序が異なります。
  ?include<<ファイル名>>の場合は下記の(2)から開始されます。
  (1) アセンブル中のソースファイルにあるディレクトリが検索されます。
  (2) アセンブラ起動時に、-Iオプションで指定したディレクトリが検索されます。複数指定した場合は、指定順に検索されます。
  (3) 標準ディレクトリが検索されます。標準ディレクトリとは、以下のディレクトリです。
    - 環境変数THOME900を設定している場合は、$THOME900/includeディレクトリです。
    - 環境変数THOME900を設定していない場合は、環境変数THOME/includeディレクトリです。
    - いずれも設定していない場合は、カレントディレクトリの下のincludeディレクトリです。
    - MS-DOS版で、環境変数THOME900またはTHOMEの設定がドライブ指定のみの場合は、そのドライブの¥includeディレクトリが検索されます。

【例】 `tumpl -I/usr/common -I../headers usr/a900/filename.mac`

というオプションを指定した場合に、どのようにファイルが探索されるかを例示します。いずれの場合も、目的のファイルが見つかった時点で探索は終了します。環境変数は次の様に設定されていると仮定します。

```
THOME900=/commom/asmsys/tlcs900
THOME=/common/all
```

`?include "def.h"`

(1) usr/a900/def.h (ソースファイルのあるディレクトリ)
(2) /usr/common/def.h (/usr/commonディレクトリ)
(3) ../header/def.h (../headersディレクトリ)
(4) 標準のディレクトリ
  (1) /commom/asmsys/tlcs900/include/def.h
  (2) /commom/all/include/def.h

の順でdef.hファイルがあるかどうかが探索されます。

`?include "sys/def.h"`

(1) usr/a900/sys/def.h (ソースファイルのあるディレクトリ)
(2) /usr/common/sys/def.h (/usr/commonディレクトリ)
(3) ../header/sys/def.h (../headersディレクトリ)
(4) 標準のディレクトリ
  (1) /commom/asmsys/tlcs900/include/sys/def.h
  (2) /commom/all/include/sys/def.h

の順でdef.hファイルがあるかどうかが探索されます。

`?include <def.h>`

(1) /usr/common/def.h (/usr/commonディレクトリ)
(2) ../header/def.h (../headersディレクトリ)
(3) 標準のディレクトリ
  (1) /commom/asmsys/tlcs900/include/def.h
  (2) /commom/all/include/def.h

の順でdef.hファイルがあるかどうかが探索されます。

`?include "/usr/include/def.h"`

ファイル名が絶対パスで指定されているため、-Iオプションを指定しても読込むファイルのディレクトリを変更することはできません。

第6章 注意事項

6.1 マクロプロセッサとアセンブラプリプロセッサの関係

システムの標準構成

アセンブラシステムはソースファイルをマクロプロセッサ(TUMPL),アセンブラプリプロセッサ(TUAPP),アセンブラの順に処理することを前提にシステムが構成されています。

マクロプロセッサ(TUMPL)とアセンブラプリプロセッサ(TUAPP)は共に条件アセンブリ機能,外部ファイルの取込み,マクロ展開(文字列の置換)機能を備えています。また、アセンブラにも外部ファイルの取込み機能があります。

マクロプロセッサ,アセンブラプリプロセッサ,アセンブラの全てを組合わせて利用できますが、次の基準で使用することをお薦めします。

- 単純な外部ファイルの取込み機能のみが必要な場合はアセンブラのみを使用して下さい。
- 単純な文字列の置換,条件アセンブルを行いたい場合はアセンブラプリプロセッサとアセンブラの組合わせて使用することをお薦めします。
- マクロプロセッサには、マクロライブラリの取込み機能,複雑なマクロ展開機能がありますので、高度なマクロ処理を使用する場合はマクロプロセッサとアセンブラを組合わせて使用して下さい。

第1章 プリプロセッサの概要

アセンブラプリプロセッサは条件アセンブル,外部ファイルの取込み,マクロ置換え(文字列の置換)等のプリプロセッサ機能を提供します。
アセンブラプリプロセッサは C コンパイラのプリプロセッサとほぼ同等の機能を備えています。

第2章 プリプロセッサ指令

プリプロセッサ機能を利用するためには、ソースプログラム中にプリプロセッサ指令を記述します。

【構文】

```
プロセッサトークン　改行
　　または
制御行　改行
　制御行は下記のいずれか
　　　# if　　定数式
　　　# ifdef　　識別子
　　　# ifndef　　識別子
　　　# elif　　定数式
　　　# else
　　　# endif
　　　# include　　プリプロセッサトークン
　　　# define　　識別子　置換リスト
　　　# define　　識別子(識別子リスト)　置換リスト
　　　# undef　　識別子
　　　#
注意：識別子と（の間には空白が入らない
　　置換リストはプリプロセッサトークン
```

【説明】 プリプロセッサ処理は、ソースファイルを行単位で扱います。制御行は行頭の"#"記号からその行の終りまでです。"#" の前後に空白文字またはタブ文字があってもかまいません。

行末に逆スラッシュ "¥" がある場合は、それに続く改行文字は空白文字とみなされ、行の連結が行われます。

【意味】 プリプロセッサによって、ソースファイルの条件アセンブルやマクロの定義、展開を行なうことができます。プリプロセッサ処理の終ったソースファイルがアセンブル対象となります。プリプロセッサ指令の処理前に、行末の逆スラッシュ "¥" による行の連結が行なわれます。したがって、プリプロセッサ指令を複数行にまたがって書く場合には各行の終りに逆スラッシュを置かなければなりません。

2.1 条件組込み

#if, #ifdef または #ifndef から対応する #endif までの間にあるトークン列は条件コンパイルによって最終的なプログラムに含まれるかどうかが決定されます。

【構文】

条件指令	if部	elif部	else部	#endif
if部	# if	定数式		グループ
	# ifdef	識別子		グループ
	# ifndef	識別子		グループ
elif部	# elif	定数式		グループ
else部	# else			グループ

【説明】 グループはif部,elif部,else部の次の行から始まる行単位のプログラムです。条件指令が入れ子になってもかまいません。
elif 部および else 部はなくてもかまいません。

【例】

```
# ifdef MAIN
# define  EXTERN
# else
# define  EXTERN  extern
# endif
```

【意味】 #ifおよび#elifでは定数式が真(0以外)と評価された場合に条件成立となります。#ifdefでは識別子がマクロ定義されている場合、#ifndefでは識別子がマクロ定義されていない場合に条件成立となります。
#if および#elif における定数式には、本説明書「第 1 部 アセンブリ言語」の「2.3.5 絶対式」で定義されている絶対式（定数式）を記述することができます。また、#ifdef および#ifndef における識別子には、本説明書「第 1 部 アセンブリ言語」の「2.2 識別子」で定義されている識別子を記述することができます。但し、次の制約があります。

- 使用できる識別子はマクロ定義されたものに限られます。プログラム中の変数名は使用できません(マクロ名と見なされます)。

#if,#ifdef,#ifndef,#elif で条件成立となった場合、対応する#else または#endif までのプログラムが組込まれます。条件不成立の場合、対応する#else または#elif から#endif までが組込まれます。
条件によって組込まれずに読み飛ばされることになったプログラムの部分は、入れ子になった#if-#else-#endif の対応をとるためだけに検査されます。

2.2 ソースファイル組込み

#include 指令によってプログラム中のその場所に別のファイルを組込むことができます。

【構文】

```
# include   <ファイル名>
# include   "ファイル名"
```

【説明】 #include指令の引数はまずマクロ展開されます。その結果ファイル名一つだけが残らなければなりません。文字列の場合のような連結は処理されません。

【意味】 #include指令によって引数に指定された文字列はファイル名と見なされ、プログラム中のその場所にファイルの内容が組込まれます。#includeされたファイル中でさらに#includeを使用することもできます。

引数の指定の形式は二種類あり、ファイル探索の行なわれるディレクトリが異なります。ソースファイルの探索されるディレクトリの詳細については「C,C-Like, アセンブラ取扱説明書[操作編]」の-I オプションの説明を参照してください。

2.3 マクロ置換え

マクロ定義

#define 指令によってプリプロセッサマクロを定義することができます。プリプロセッサマクロにはオブジェクト形式マクロと関数形式マクロがあります。関数形式マクロは引数をとることができます。

【構文】

```
# define 識別子 置換リスト
# define 識別子 ( 識別子リスト ) 置換リスト
  注意：識別子と ( の間は空けてはいけません
```

【説明】 上の構文で一行目がオブジェクト形式マクロ定義、二行目が関数形式マクロ定義となります。

関数形式マクロ定義では、マクロ名となる識別子と左かっこ "(" の間に空白文字、コメントなどがあってはなりません。関数形式マクロの識別子リストには、仮引数となる識別子をコンマ "," で区切って書きます。

置換リストの前後にある空白類文字は置換リストの一部とは見なされません。

【例】

```
/* オブジェクト形式マクロ */
#define NAMELEN 128
#define global extern

/* 関数形式マクロ */
#define bcopy(s, d, n) memcpy(d, s, n)
```

【意味】 #defineの次に記述される識別子がマクロ名となります。プリプロセッサマクロはコンパイル時定数に名前を与えたり、関数の別名や簡単な関数のインライン定義を与えたりすることができます。

マクロ定義は、対応する#undef 指令まで、または翻訳単位の終りまで有効です。

マクロを使ってプリプロセッサ指令を生成することはできません。行頭の "#" に続く識別子は同名のマクロがある場合でもプリプロセッサ指令と見なされ展開はされません。また、マクロ展開の結果がプリプロセッサ指令と同じ文字列になったとしても、それはプリプロセッサでは処理されません。

マクロは再定義されるとエラーとなります。ただし、マクロ名および置換リストの内容、仮引数の名前が全て同じ定義の場合にはエラーになりません。

2.4 マクロ定義の無効化

【構文】

マクロ定義の無効化

undef 識別子

【説明】 識別子で指定された名前のマクロがすでに定義されている場合にそのマクロの定義を無効にします。指定されたマクロが定義されていない場合はこの指令は無視されます。

【例】

```
/* マクロ定義 */
#define DEBUG_LEVEL 0
#define bcopy(s, d, n) memcpy(d, s, n)
/* マクロ定義の無効化 */
#ifdef DEBUG
#undef DEBUG_LEVEL
#define DEBUG_LEVEL 1
#endif
#undef bcopy
#define bcopy(s, d)  memcpy(d, s, strlen(s))
#undef touppe
```

【意味】 オブジェクト形式、関数形式のどちらのマクロ定義でも、マクロ名を指定することでマクロ定義を無効にすることができます。

すでに定義されているマクロを再定義する場合には、再定義に先立ち、#undef 指令でマクロ定義を無効にしておく必要があります。

2.5 マクロ展開

ソースプログラム中にマクロ名が出現した場合、それはマクロ呼出しと見なされ展開されます。ただし、文字定数および文字列リテラルの中にあるマクロは展開されません。
関数形式マクロの呼出しは以下の手順で処理されます。

実引数の判定

関数形式マクロの呼出しは関数呼出しと構文上同じ形をしています。各実引数はコンマ "," で区切られますが、対応するかっこ内のコンマは実引数の区切りにはなりません。マクロ呼出しの最も外側のかっこおよび実引数区切りのコンマは実引数の一部とはなりません。
実引数が空である場合、ワーニングとなります。
また、プリプロセッサ指令を含むこともできません。
関数形式マクロの呼出しは複数行にまたがることができます。このとき改行文字は通常の空白類文字として扱われます。
空の実引数が指定された場合(例えば A( ) など)、マクロ定義の仮引数に前処理字句が無い場合はエラーとはなりませんが、仮引数に前処理字句がある場合はワーニングとなります。

実引数内のマクロの展開

実引数はマクロの置換リスト内の仮引数と置換えられる前に、それぞれが独立したプログラムテキストであるかのようにマクロ展開されます。

マクロの展開

マクロ呼出しを置換リストで置換えます。このとき各仮引数は対応する実引数で置換えられます。ただし、文字定数、文字列リテラルの中にある仮引数は置換えの対象になりません。
置換えられた結果はさらにマクロ呼出しを含むかどうか再走査されます。

【例】

```
#define ASIZE  100
#define init()  ld sp,0x8000
#define MAXSIZE  MAXARRAY
#define MAXARRAY  100
```

上記のマクロ定義がある場合のマクロ展開の例を示します。

```
ld  ix,ASIZE     →  ld ix,100
init()           →  ld sp,0x8000
ld xix,MAXSIZE   →  ld xix,MAXARRAY
                 →  ld xix,100
```

最後の例では、MAXSIZE が MAXARRAY に置換され、さらに再走査で MAXARRAY が 100 に置換されました。

2.6 再走査と再置換え

マクロ置換えの後、結果として得られたトークンの列は、さらにその中にマクロを含むかどうか再走査されます。このとき、ソースファイル上で続いているプリプロセッサトークンも考慮に入れられます。
マクロ置換えの結果生じた識別子が置換え中のマクロと同じ名前である場合、その識別子はそれ以上マクロ展開されません。再帰的なマクロが展開された場合、同じマクロは二度展開されることはありません。

【例】 次のようなマクロを定義した場合、展開結果の"foo"はこれ以上展開されません。

```
/* マクロ定義 */
#define foo(x, y) foo((x)+1, (y)+1)
/* マクロ呼出 */
foo(a, 9)
/* マクロ展開 */
foo((a)+1, (9)+1)    /* この結果の"foo"はこれ以上展開されない */
```

さらにマクロ呼出しが重なった場合は次のようにエラーとなります。

```
/* マクロ定義 */
#define A B
#define B C(x)
#define C(x) A(x+1)      /* Macro C recursion error */
```

2.7 空指令

#と空白類文字だけから成る行は空指令と呼ばれ、無視されます。

第3章 展開処理

3.1 コメント

プリプロセッサで処理するコメントには 2 種類あります。一つは C 言語と同様の "/*" と "*/" で囲まれた部分、もう一つはプリプロセッサ文の後に記述された "#" に続く部分です。これらはアセンブリ言語のコメント記号 ";" と異なりますので、コーディング時には注意が必要です。
アセンブラプリプロセッサは一行単位の処理を基本としていますので、コメントを記述する場合も一行で終了するようにした方がよいでしょう。

```
#define TRUE 0xff          # ここはコメント
#define FALSE 0x00         /* ここはコメント */
#define START 0x8000       /* ここはコメント
                              ここもコメント */
; ここはアセンブラで処理されるコメント
```

3.2 展開後のプリプロセッサ行

プリプロセッサがプリプロセッサ行を展開した後は、その行は空白行になります。したがって、プリプロセッサ文に付随するコメント部も出力ファイルに展開されません。

第4章 注意事項

4.1 アセンブラプリプロセッサとマクロプロセッサの関係

システムの標準構成

アセンブラシステムはソースファイルをマクロプロセッサ(TUMPL),アセンブラプリプロセッサ(TUAPP),アセンブラの順に処理することを前提にシステムが構成されています。

マクロプロセッサ(TUMPL)とアセンブラプリプロセッサ(TUAPP)は共に条件アセンブリ機能,外部ファイルの取込み,マクロ展開(文字列の置換)機能を備えています。また、アセンブラにも外部ファイルの取込み機能があります。

マクロプロセッサ,アセンブラプリプロセッサ,アセンブラの全てを組合わせて利用できますが、次の基準で使用することをお薦めします。

- 単純な外部ファイルの取込み機能のみが必要な場合はアセンブラのみを使用して下さい。
- 単純な文字列の置換,条件アセンブルを行いたい場合はアセンブラプリプロセッサとアセンブラの組合わせて使用することをお勧めします。
- マクロプロセッサには、マクロライブラリの取込み機能,複雑なマクロ展開機能がありますので、高度なマクロ処理を使用する場合はマクロプロセッサとアセンブラを組合わせて使用して下さい。

付録 A 修正履歴

本マニュアルの主要な修正の履歴は次のとおりです。

版	発行日	修正点
初版	1993年4月1日	初版
第2版	1993年10月1日	誤字,脱字の修正
		符号無し整数式の値の有効範囲を変更
第3版	1995年4月1日	浮動小数点型定数サポートによる仕様追加
第4版	1996年4月1日	誤字,脱字の修正､説明の追加,修正
第5版	1997年4月1日	リンケージエディタコマンドファイルの拡張子を.lcfに変更
第7版	1999年10月1日	TX19アセンブラ固有、TX19リンカ固有部分追加